

# Chapter 2

## Basic Graph Algorithms

### 2.1 A Brief Introduction to Graph Theory

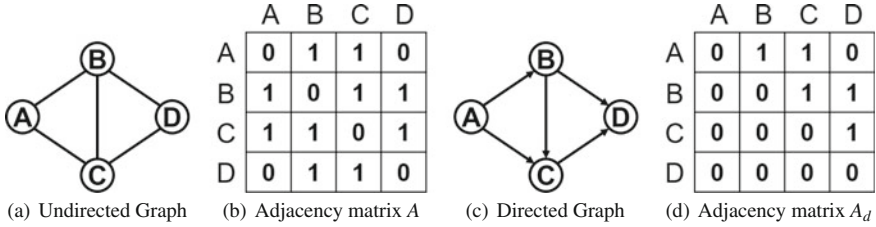
A graph  $G(N, E)$  consists of two finite sets  $N$  and  $E$ . The elements of  $N$  are called nodes and the elements of  $E$  are called edges of  $G$ . If an edge  $e(a, b)$  joins nodes  $a$  and  $b$ , then  $a$  and  $b$  are incident (or adjacent) with edge  $e(a, b)$ . A directed edge is an edge where one incident node (or endpoint) is designated as the tail and the other incident node (or endpoint) is designated as the head. A directed graph (or digraph) is a graph each of whose edges is directed.

A line graph  $L(G)$  of  $G$  is a graph where each node of  $L(G)$  represents an edge of  $G$  and two nodes of  $L(G)$  are adjacent if and only if they are adjacent as edges in  $G$ . A bipartite graph  $G(N, E)$  is a graph whose node-set  $N$  can be partitioned into two subsets  $N_1$  and  $N_2$  such that each edge of  $G$  has one endpoint in  $N_1$  and one endpoint in  $N_2$ . A path  $P_k$  is a non-empty graph  $P_k(N, E)$  of the form  $N = n_1, n_2, \dots, n_k$  and  $E = e(n_1, n_2), e(n_2, n_3), \dots, e(n_{k-1}, n_k)$  where all nodes are all distinct. Two or more paths are independent if none of them contains the inner nodes ( $n \in N - n_1, n_2$ ) of another. The distance  $d(s, t)$  in  $G$  is the length of a shortest path from  $s$  to  $t$ .

A node-cut in a connected graph  $G$  is a node-set  $C$  whose removal disconnects the graph  $G$ . A cut-node (or articulation node) is a node-cut consisting of a single node. Similarly, an edge-cut in a connected graph  $G$  is an edge-set  $D$  whose removal disconnects the graph  $G$ . A cut-edge is an edge-cut consisting of a single edge.

### 2.2 Network Representations

There are many ways to represent a network structure. In this section, we present the most common and well-known data structures for network representation.



**Fig. 2.1** Node-node adjacency matrix representation

### 2.2.1 Node-Node Adjacency Matrix

The node-node adjacency matrix  $A$  of an undirected graph  $G$  is a symmetric matrix whose rows and columns are defined as

$$A[i, j] = \begin{cases} 1 & \text{if node } i \text{ and node } j \text{ are adjacent} \\ 0 & \text{otherwise} \end{cases}$$

The node-node adjacency matrix can assign a weight (e.g., cost or capacity) of edges  $(i, j)$  on the  $ij$ th element in the matrix. Clearly,  $A$  is a symmetric matrix with zeros on the diagonal and only the upper (or lower)-triangular part of the matrix is stored. The sum of the elements in any  $i$ th row (or  $j$ th column) of the matrix is the degree of node  $i$  (or  $j$ ). The power of  $A$  (i.e.,  $A^k$ ) is the number of walks of length  $k$  from  $i$  to  $j$ . Figure 2.1b shows an example of the node-node adjacency matrix ( $A$ ) of the undirected graph in Fig. 2.1a.

The node-node adjacency matrix  $A_d$  of a directed graph  $G$  is a matrix whose rows and columns are defined as

$$A_d[i, j] = \begin{cases} 1 & \text{if there is a directed edge from node } i \text{ to node } j \\ 0 & \text{otherwise} \end{cases}$$

The sum of the elements of the  $i$ th row of the matrix is the out-degree of node  $i$  and the sum of the elements of the  $j$ th column of the matrix is the in-degree of node  $j$ . Figure 2.1d shows an example of the node-node adjacency matrix ( $A_d$ ) of the directed graph in Fig. 2.1c. Let  $n$  be the number of nodes. The total space requirement of the node-node adjacency matrix is  $O(n^2)$ , which is inefficient if the graph is sparse.

### 2.2.2 Node-Edge Incidence Matrix

The node-edge incidence matrix  $I$  of an undirected graph  $G$  is a matrix whose rows and columns are defined as

**Fig. 2.2** Node-edge incidence matrix representation

	AB	AC	BC	BD	CD
A	1	1	0	0	0
B	1	0	1	1	0
C	0	1	1	0	1
D	0	0	0	1	1

(a) Incidence Matrix  $I$

	AB	AC	BC	BD	CD
A	1	1	0	0	0
B	-1	0	1	1	0
C	0	-1	-1	0	1
D	0	0	0	-1	-1

(b) Incidence Matrix  $I_d$

$$I[i, j] = \begin{cases} 1 & \text{if node } i \text{ is incident to edge } j \\ 0 & \text{otherwise} \end{cases}$$

The sum of the elements in the  $i$ th row of the matrix is the degree of node  $i$  and the sum of the elements in any column of the matrix is 2. Figure 2.2a shows an example of the node-edge incident matrix ( $I$ ) of the undirected graph in Fig. 2.1a.

The incidence matrix  $I_d$  of an directed graph  $G$  is a matrix whose rows and columns are defined as

$$I_d[i, j] = \begin{cases} -1 & \text{if node } i \text{ is the tail of edge } j \\ 1 & \text{if node } i \text{ is the head of edge } j \\ 0 & \text{otherwise} \end{cases}$$

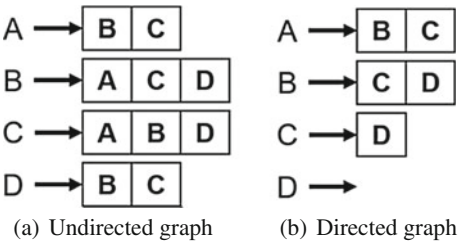
The node-edge incidence matrix  $I_d[i, j]$  represents the orientation of a directed graph  $G$ . Figure 2.2b shows an example of the node-edge incidence matrix ( $I_d$ ) of the directed graph in Fig. 2.1c. Let  $n$  be the number of nodes and let  $m$  be the number of edges. The total space requirement of the node-node adjacency matrix is  $O(n \cdot m)$ , which is also inefficient if the graph is sparse.

### 2.2.3 Adjacency List

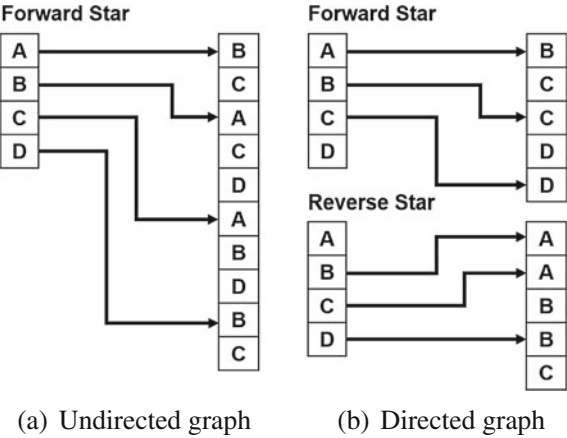
Both the incidence matrix and adjacency matrix are inefficient for sparse graphs because most of the elements in the matrix are zeros. To remedy this issue, the adjacency-list representation of a graph  $G$  uses a linked-list (or array) to represent a set of incident nodes (or edges) of the node. Figure 2.3 shows examples of the adjacency list representation corresponding to the examples in Fig. 2.1a, c.

The adjacent list is more space-efficient than the adjacency (or incidence) matrix. However, it takes linear time to test whether a node (or edge) is incident to node  $n$  whereas both the node-node adjacency matrix and node-edge incident matrix take constant time.

**Fig. 2.3** Adjacency list representation



**Fig. 2.4** Forward star representation



**2.2.4 Forward Star**

The forward star representation is similar to the adjacency list representation. It uses a single array to store a set of incident nodes (or edges) and maintains a pointer for each node that indicates the start address of a set of incidents. For a directed graph, we construct the forward star to represent the outgoing edges for each node and the reverse star to represent the incoming edges for each node. Figure 2.4 shows forward star (or reverse star) representations corresponding to the examples in Fig. 2.1a, c.

The forward star representation is more space-efficient than the adjacent list representation. However, it is more difficult to update the graph structure whereas the adjacency list representation requires constant time for addition and linear-time for update and deletion.

**2.3 Shortest Paths**

This section reviews the computational methods for finding shortest paths between nodes on a weighted graph. The shortest path computation is the most important component in developing spatial network queries including nearest neighbors, distance

estimation, the shortest route, etc. It also serves as a building block for developing more complex network queries. For example, the max-flow computation can be solved by iteratively finding a shortest path on the residual network (see Sect. 2.5).

2.3.1 Single-Source Shortest Path (SSSP)

Given a directed graph  $G(N, E)$  where every edge  $e \in E$  has a weight (or cost)  $c(e)$ , the Single-Source Shortest Path (SSSP) problem is to find a shortest simple path from a give source node  $s$  to every other nodes  $n \in N$ . If the graph includes a negative-weight cycle, the problem is NP-complete [12]. If no negative-weight cycle exists, there is a polynomial-time algorithm [2, 5].

The shortest path algorithms can be divided into two major groups: (1) Label-Correcting (LC) methods and (2) Label-Setting (LS) methods. Both approaches maintain a distance label  $d(n \in N)$  for every node, which serves as an upper bound on the shortest path distance to node  $n$ . The Label-Correcting (LC) method can solve the shortest path problem in the general case in which the edge weights may be negative. It uses local information (i.e., incident nodes) of every node and updates (or reduces) the distance label (i.e.,  $d(n)$ ) of every node at each iteration. If no negative-weight cycle exists, the LC method finds the optimal solution in polynomial time. Examples of LC method include the Bellman-Ford-Moore algorithm [4, 11, 18]. Figure 2.5b shows an example of each step of the Bellman-Ford-Moore algorithm. Figure 2.5a illustrates the input with a transportation network (four nodes and five edges). Every edge is associated with a distance (e.g., travel time), as indicated by the number displayed alongside it. Assume that node A is a source node of  $G$ . First, the Bellman-Ford-Moore algorithm initializes the distance label of the source node ( $d_1(A)$ ) to 0 and all others to  $\infty$ . Let the predecessor of  $n$  be  $n_{pre}$ . Then, the algorithm iteratively decreases  $d_{i+1}(n)$  according to the following condition.

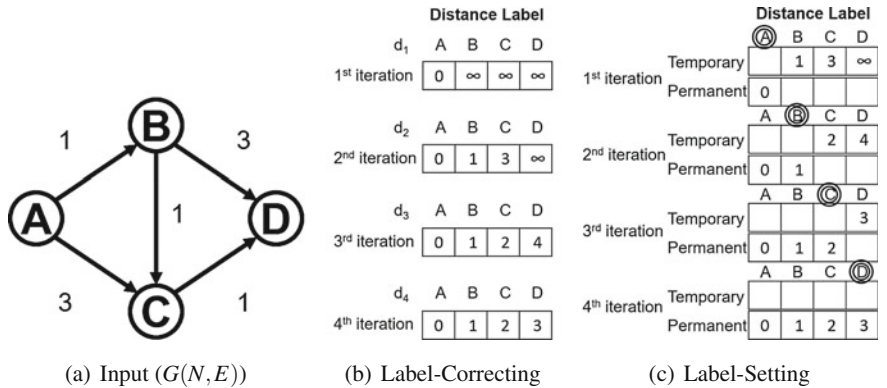


Fig. 2.5 Examples of Single-Source Shortest Path algorithms

$$d_{i+1}(n) = \min(d_i(n), d_i(n_{pre}) + c(n_{pre}, n)), \quad (2.1)$$

where  $c(n_{pre}, n)$  is the edge weight (or cost) of the edge from node  $n_{pre}$  to node  $n$ .

In the example in Fig. 2.5b, the second iteration updates the distance labels of all nodes as follows:

- $d_2(A) : \min(d_1(A)) = 0$
- $d_2(B) : \min(d_1(B), d_1(A) + c(AB)) = 1$
- $d_2(C) : \min(d_1(C), d_1(A) + c(AC), d_1(B) + c(BC)) = 3$
- $d_2(D) : \min(d_1(D), d_1(B) + c(BD), d_1(C) + c(CD)) = \infty$

This process continues until no further update is possible. The fourth iteration shows the shortest path distance from  $s$  to other nodes. The time complexity of the Bellman-Ford-Moore algorithm is  $O(n \cdot m)$ , where  $n$  is the number of nodes and  $m$  is the number of edges of the graph.

The Label-Setting (LS) method can solve the shortest path problem if the edge weights are non-negative. It maintains temporary and permanent distance labels on every node. The basic idea of the algorithm is to fan out from source node  $s$  and permanently label other nodes in the order of their distances from node  $s$ . The algorithm selects a node  $n \in N$  with the minimum temporary label, makes it permanent, and scans out-going edges of node  $n$  to update (or reduce) the temporary distance labels of incident nodes. The algorithm terminates when all nodes are permanently labeled. The output of the LS method is a shortest-path tree. It grows a shortest-path tree, starting at node  $s$ , by adding an outgoing-edge whose endpoint is as close as possible to  $s$  at each iteration. Examples of LS methods include Dijkstra's algorithm and the  $A^*$  algorithm [8, 13]. Figure 2.5c shows an example of each step of Dijkstra's algorithm. First, Dijkstra's algorithm selects the source node  $A$ , permanently labels the distance of  $A$ , and updates the temporary distance labels of  $B$  and  $C$ . Then, it selects the node that has the minimum temporal distance label (i.e.,  $B$ ), permanently labels the distance of  $B$ , and update the temporary distance labels of  $C$  and  $D$ . This process continues until all nodes are permanently labeled. The forth iteration shows the shortest path distance from  $s$  to other nodes. The time complexity of Dijkstra's algorithm is  $O(m + n \cdot \log n)$ , where  $n$  is the number of nodes and  $m$  is the number of edges of the graph [20]. Dijkstra's algorithm is faster than Bellman-Ford-Moore's algorithm but it can not produce the optimal solution when a negative weighted edge exists in the graph.

### 2.3.2 All-Pairs Shortest Paths (APSP)

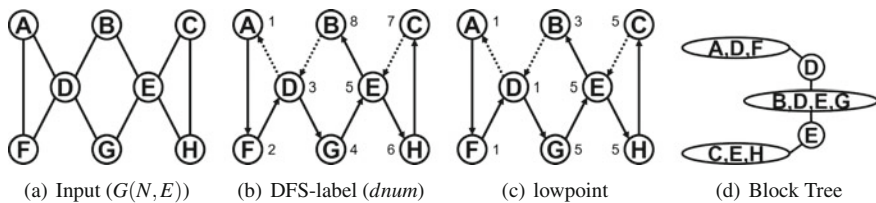
Given a directed graph  $G(N, E)$  where every edge  $e \in E$  has a weight (or cost)  $c(e)$ , the All-Pairs Shortest Paths (APSP) problem is to find the shortest path distance between all pairs of nodes. The most popular and well-known method for the APSP problem is the Floyd-Warshall algorithm [10, 23]. The Floyd-Warshall algorithm creates a node-node adjacency matrix and uses the concept of dynamic programming to

find all-pairs shortest paths on a graph. It runs in  $O(n^3)$  time, which is asymptotically no better than  $n$  calls to the SSSP algorithm from each node. For a sparse graph with non-negative edge weights, we could solve the all-pairs shortest paths by calling Dijkstra's algorithm from each node. This gives us an  $O(m \cdot n + n^2 \cdot \log n)$  algorithm [20]. If the graph contains negative-weight edges, we can re-write the graph using the node potential function such that all edge weights are non-negative. The details about this function are present in Sect. 2.6. This leads us to apply Dijkstra's algorithm to the APSP problem with negative-weighted edges [14].

## 2.4 Block Decomposition

Spatial network queries for connectivity are important to measure the robustness and resilience of network structures. The vulnerable points (or cut points) in a network can be identified efficiently through a block decomposition technique. A graph  $G = (N, E)$  is connected if any two nodes in  $G$  are linked by a path. A connected graph  $G$  is called bi-connected if for every node  $n \in N$ ,  $G - n$  is connected. An articulation node of a graph  $G$  is a node  $n$  such that  $G - n$  is disconnected. A block of a graph is a maximal connected sub-graph  $H$  such that no node of  $H$  is an articulation node of  $H$  [7]. Consider the example in Fig. 2.6a. The removal of node  $D$  makes graph  $G(N, E)$  disconnected. We can see that there are two articulation nodes (i.e.,  $D$  and  $E$ ). A block is a maximal connected subgraph without an articulation node. Graph  $G$  can be decomposed into these blocks which can show the overall structure of  $G$  using a tree.

Figure 2.6 shows examples of each step of the Depth-First-Search (DFS)-based algorithm [22]. First, it performs a Depth-First-Search (DFS), starting from an arbitrary node in  $G$ , and labeling nodes in the order in which they are discovered. We refer to this label as a *dnum*. Figure 2.6b shows the *dnum* for every node. The edges of the original graph can be divided into two types: tree edges and back edges. A tree edge belongs to the DFS-spanning tree itself; it connects a node to one of its descendants whereas a back edge connects a node to one of its ancestors. Let  $e$  be a back edge whose endpoints are  $a$  and  $b$ . Then  $dnum(a) < dnum(b)$ . Figure 2.6b shows tree edges (solid lines) and back edges (dotted lines) based on the original graph. Let tree  $T$  be the output of applying DFS to  $G$ . Then a non-root node  $n$  of  $T$  is



**Fig. 2.6** Block decomposition

an articulation node of  $G$  if and only if node  $n$  has a child  $n_c$  such that no descendant of  $n_c$  has a path to an ancestor of node  $n$  by a back edge.

Let a set of tree edges be  $TE$  and let a set of back edges be  $BE$ . Let us define  $lowpoint(a)$  as following.

$$lowpoint(a) = \min((lowpoint(b) \text{ where } ba \in TE) \cup (dnum(b) \text{ where } ab \in BE))$$

Figure 2.6c shows the  $lowpoint$  for every node. Given a tree edge  $ab$ , we can test whether node  $a$  is an articulation node as shown by the following condition: node  $a$  is an articulation node if and only if  $dnum(a) \leq lowpoint(b)$  or node  $a$  is on at least two tree edges. Consider the example in Fig. 2.6. We can easily find two articulation nodes (i.e.,  $D$  and  $E$ ) because  $dnum(D) < lowpoint(G)$  and  $dnum(E) = lowpoint(H)$ . Figure 2.6d shows a block decomposition containing three blocks.

## 2.5 Maximum Network Flow

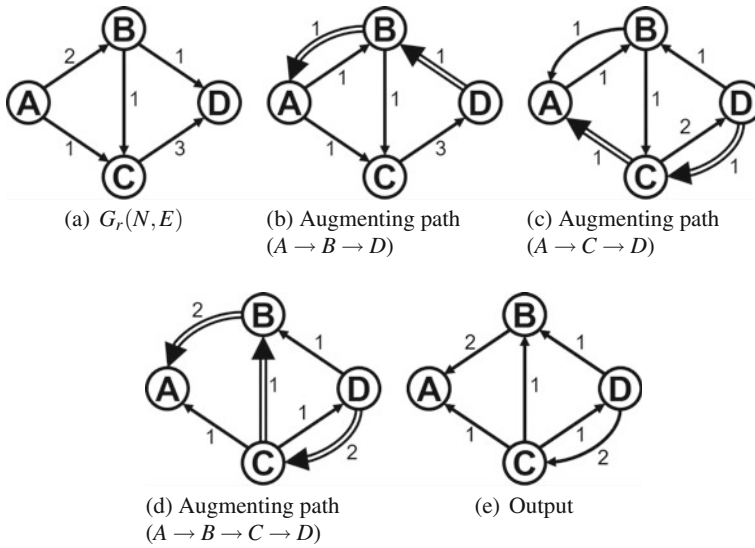
Given a graph  $G(N, E)$ , a set of capacity constraints on edges  $c(e \in E) \in C$ , a source node  $s$ , and a sink node  $t$ , the maximum network flow problem is to find a flow of maximum value that meets capacity constraints. This general problem arises in many real applications (e.g., network flow analysis) and efficient algorithms for computing the maximum flow are critical for handling large-sized spatial networks. This section presents two general methods for solving the maximum-flow problem: (1) the Augmenting-Path (AP) method and (2) the Push-Relabel (PR) method.

### 2.5.1 Augmenting-Path Algorithm

The Augmenting-Path (AP) method is based on the idea of a residual network and augmenting paths [2]. A residual network represents the available capacities ( $r \in R$ ) on the current network flow  $f$ . Let  $c(e)$  be a capacity on edge  $e$  and let  $f(e)$  be a flow on edge  $e$ . The residual capacity  $r(e)$  on edge  $e$  can be defined as follows

$$r(e) = \begin{cases} c(e) - f(e) & \text{if } e \text{ is a forward edge} \\ f(e) & \text{if } e \text{ is a backward edge} \end{cases}$$

An augmenting path  $p$  is a simple path from  $s$  to  $t$  in the residual network  $G_r(N, E, R)$ . The AP method repeatedly identifies an augmenting path from  $s$  to  $t$  according to the available capacities in the residual network, then adds the path to the flow, and updates the capacities in the residual network. This process continues until no augmenting path can be found. It can be shown that the flow through



**Fig. 2.7** Augmenting Path algorithm

a network is optimal if and only if it contains no augmenting path on the residual network [2].

Let  $f$  be a flow in  $G$ . A feasible (or admissible) flow  $f$  in  $G$  satisfies the following two conditions:

1. Capacity Constraint:  $f(e) \leq c(e)$ , for every edge  $e$  in  $G$ .
2. Conservation Constraint:  $\sum_{e \in In(n)} f(e) = \sum_{e \in Out(n)} f(e)$  for every node  $n \in N - s, t$ , where  $In(n)$  is a set of incoming edges of node  $n$  and  $Out(n)$  is a set of outgoing edges of node  $n$ .

Assume that  $N$  is divided into two sets,  $s \in N_1$  and  $t \in N_2$ . Let the pair  $(N_1, N_2)$  be a cut in  $G$  and  $c(N_1, N_2)$  be the capacity of this cut. It is well known that the maximum total value of a flow equals the minimum capacity of a cut [11].

Consider the example of AP method in Fig. 2.7. Figure 2.7a illustrates the residual network where every edge is associated with a capacity, as indicated by the number displayed alongside it. Given the residual network, the algorithm identifies the augmenting path on the available capacities and updates the residual network. We define the residual capacity of the augmenting path as the minimum residual capacity of any edge in the path. In each iteration, the algorithm finds an augmenting path  $p$  and updates the flow on each edge of  $p$  by the residual capacity of the augmenting path. In this example, the first iteration identifies path  $A \rightarrow B \rightarrow D$  as an augmenting path and computes the residual capacity of the path (i.e., 1) (see Fig. 2.7b). This augmentation reduces the capacity of edge  $AB$  by 1 and increases the capacity of edge  $BA$  by 1. It also reduces the capacity of edge  $BD$  by 1 and increases the capacity of edge  $DB$  by 1. Figure 2.7b shows the residual network after augmenting path  $A \rightarrow B \rightarrow D$ . After

three iterations, the algorithm achieves the optimal solution (Fig. 2.7e), augmenting path  $A \rightarrow B \rightarrow D$ , path  $A \rightarrow C \rightarrow D$ , and path  $A \rightarrow B \rightarrow C \rightarrow D$ . Figure 2.7e shows the output of the Augmenting-Path algorithm.

### 2.5.2 Push-Relabel Algorithm

The Push-Relabel (PR) method is based on the notion of pre-flow which ignores the constraint that in-flow should be equal to out-flow at every node and iteratively pushes flows from one node to other nodes [2, 5]. Let  $inflow(n)$  be the total amount of flow incoming to node  $n$  and let  $outflow(n)$  be the total amount of flow outgoing from node  $n$ . We define the excess of node  $n$  as  $inflow(n) - outflow(n)$ . We refer to a node with a positive excess as an active node. The key idea is to push as much pre-flow as possible from the active node toward the sink  $t$ . Every node has a height label which can determine the direction of pre-flow. We only push pre-flow from a higher labeled node to a lower labeled node. As an initial step, the height of the source (i.e.,  $s$ ) is labeled as the total number of nodes (i.e.,  $|N|$ ) and the heights of others are labeled as 0. The height of a node can be increased to one unit more than the height of the lowest of its incident nodes when the node cannot push all excess flow to its incident nodes. The algorithm selects an active node, pushes pre-flow from the active node until either the node's excess becomes zero or the algorithm relabels the node. The algorithm terminates when the network contains no active node.

Consider the example in Fig. 2.8. Figure 2.8a illustrates the residual network where every edge is associated with a capacity, as indicated by the number displayed alongside it. Given the residual network, the algorithm labels the heights of every node (see Fig. 2.8b). Then, it chooses node  $A$  as an active node and pushes excess flow to incident nodes (i.e.,  $B$  and  $C$ ) (see Fig. 2.8c). Figure 2.8d shows the residual network after pushing excess flow from node  $A$ .

Next, the algorithm chooses node  $B$  as an active node. Since no outgoing-edge exists for pre-flow, the height of node  $B$  increases by 1 (see Fig. 2.9b). Then, it pushes excess flow from node  $B$  to incident nodes (i.e.,  $C$  and  $D$ ) (see Fig. 2.9b). Figure 2.8d shows the residual network after pushing excess flow from node  $B$ .

After that, the algorithm chooses node  $C$  as an active node, increases the height by 1, and pushes the excess flow to node  $D$ . Since there is no excess node except the

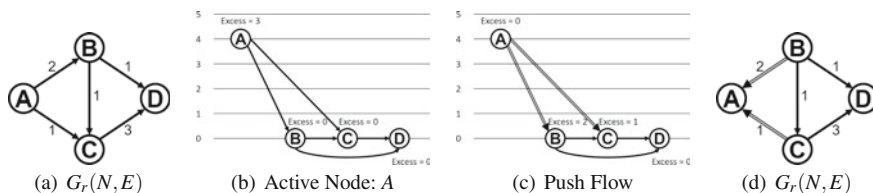


Fig. 2.8 Push Relabel algorithm: 1st iteration

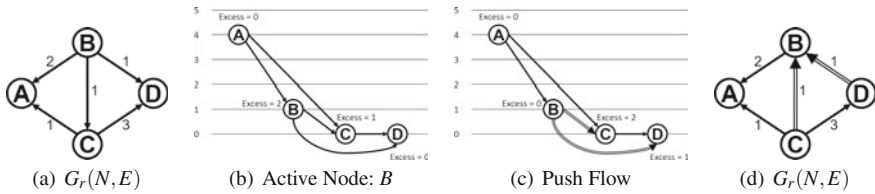


Fig. 2.9 Push Relabel algorithm: 2nd iteration

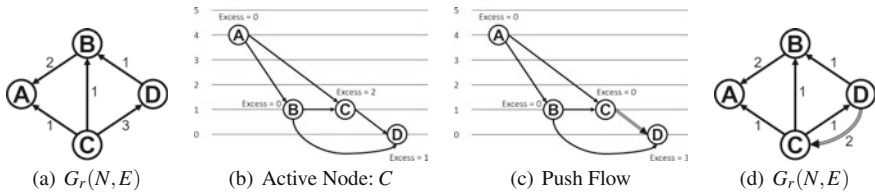


Fig. 2.10 Push Relabel algorithm: 3rd iteration

sink  $t$ , the algorithm terminates. Figure 2.10d shows the output of the Push-Relabel algorithm.

## 2.6 Bipartite Weighted Matching

Graph  $G(N, E)$  is bipartite if the nodes can be divided into two sets,  $G_1$  and  $G_2$ , such that all edges ( $e \in E$ ) have one node in  $G_1$  and one node in  $G_2$ . A matching in a graph  $G(N, E)$  is a subset of edges  $\hat{E} \subset E$  such that no two edges of  $\hat{E}$  share a node. Given a weighted bipartite network  $G = (N_1 \cup N_2, E)$  with  $|N_1| = |N_2|$  and edge weights  $w(e)$ , the Bipartite Weighted Matching problem is to find the perfect matching with the minimum total weight. The problem is important in many practical applications, such as resource assignment. In this section, we introduce the Successive Shortest Path algorithm that can produce the optimal solution for the Bipartite Weighted Matching problem [2].

Consider the resource assignment problem in Fig. 2.11. Figure 2.11a shows an example input network consisting of a graph with 4 graph-nodes ( $A, B, C, D$ ) and two service center nodes ( $X, Y$ ) with capacities of 2 each. The objective of this problem is to assign graph-nodes to a service center that minimizes the total distance between graph-nodes ( $N_1$ ) to service center nodes ( $N_2$ ). We assume that every service center node can be matched with exactly two graph-nodes. Figure 2.11b shows a bipartite graph, consisting of two sets: graph nodes ( $N_1$ ) and service center nodes ( $N_2$ ). For efficiency, we create a super-source node  $S$  that is connected to every node in  $N_1$  by an edge of weight 0 and create a sink node  $T$  and connect it to every node in  $N_2$  by an edge of weight 0. Every edge is associated with the distance from a graph-node to a service center node, as indicated by the number displayed alongside it.

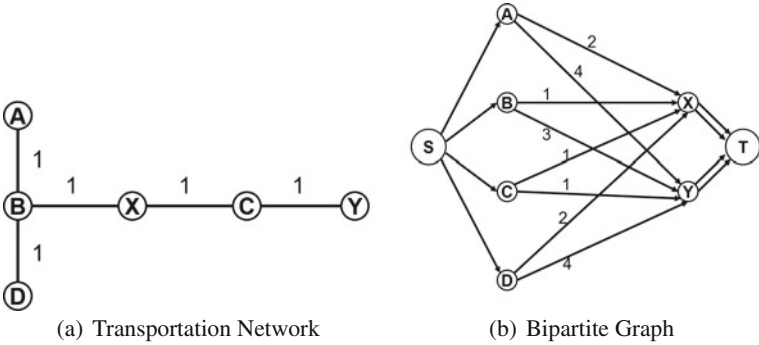


Fig. 2.11 Example of resource assignment

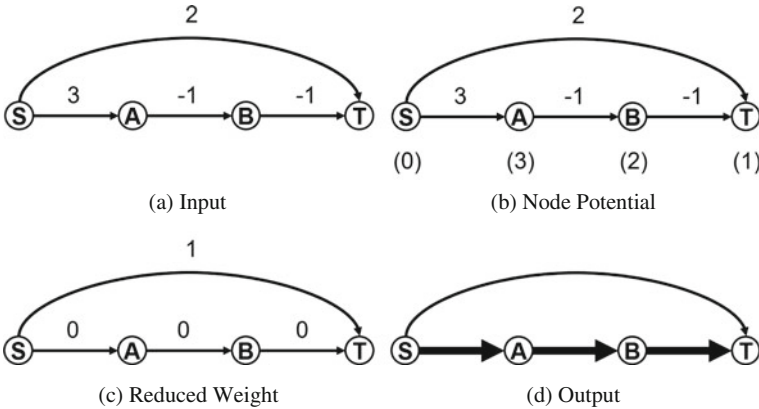


Fig. 2.12 Node potential and reduced weight

The matching problem in Fig. 2.11b is identical to the Bipartite Weighted Matching problem if each service center node can be split into two nodes. In this example, we use the Successive Shortest Path algorithm to find the optimal matching.

The Successive Shortest Path algorithm uses three important notions: node potential, reduced weight function, and shortest augmenting path. Given a weighted, directed graph  $G(N, E)$  with edge weights  $w(e)$ , let  $p(a)$  be a shortest path distance from source node  $s$  to node  $a$ . We refer to  $p(a)$  as the node potential of node  $a$ . Then the reduced weight function can be defined as follows:  $w(a, b) = w(a, b) + p(a) - p(b)$ . Consider the example in Fig. 2.12a. The shortest path from  $S$  to  $T$  is  $S \rightarrow A \rightarrow B \rightarrow T$ . We can use the Label-Correcting (LC) method to compute the shortest path in the graph with negative weighted edges, but the Label-Setting (LS) method usually outperforms the Label-Correcting (LC) method. If the reduced weight function converts all edge-weights into non-negative edge-weights, more efficient algorithms (i.e., the LS method) can be applied to find the shortest path. Figure 2.12b shows the node potential for every node and Fig. 2.12c shows the reduced weight on every edge.

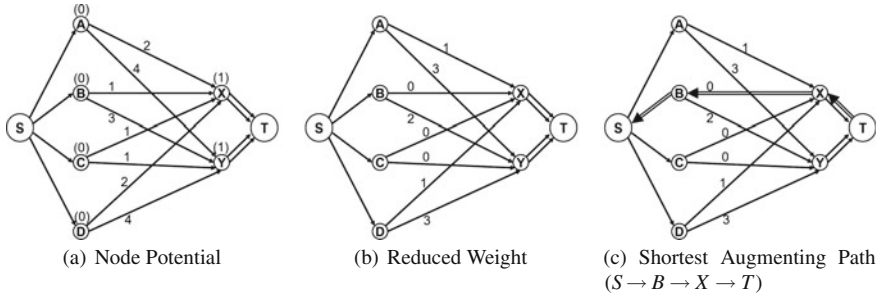


Fig. 2.13 Successive Shortest Path algorithm: 1st iteration

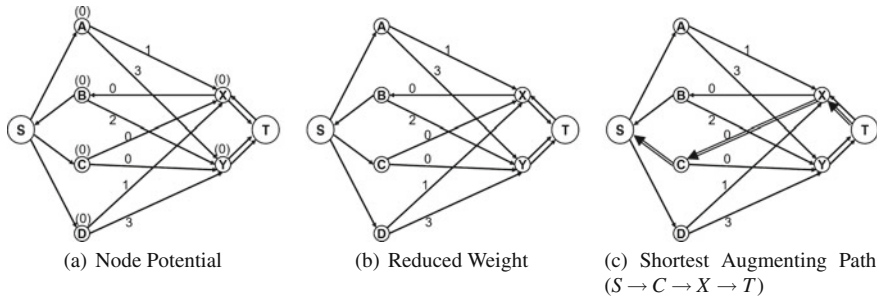
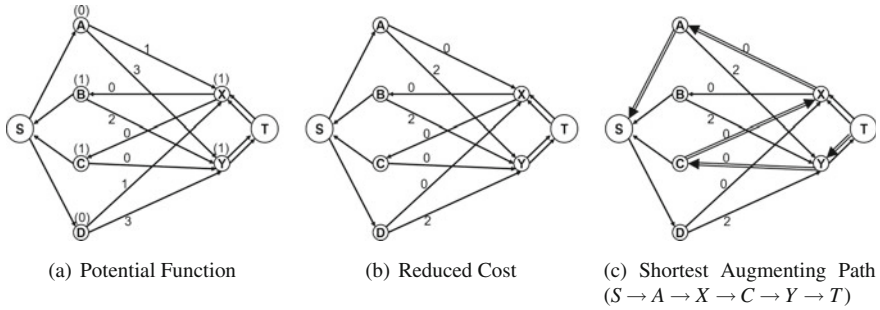


Fig. 2.14 Successive Shortest Path algorithm: 2nd iteration

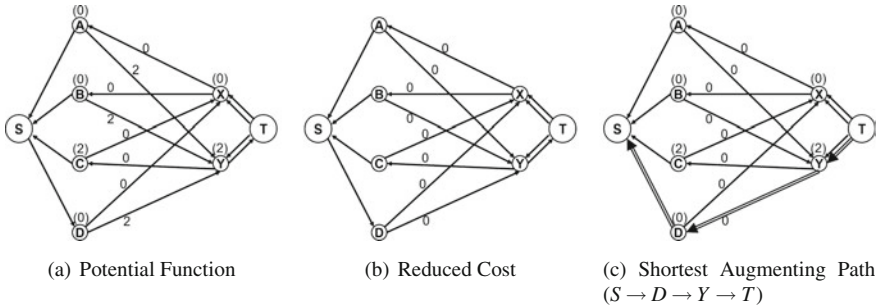
Since all edge-weights are non-negative, we can easily find the shortest path using the LS method. A node potential is a generalization of the concept of distance label that we used in the Successive Shortest Path algorithm.

Consider the example in Fig. 2.11b. We assume that every edge has a unit capacity. Figure 2.13a, b show the node potentials and reduced weights produced from Fig. 2.11b. In this example, we augment the shortest path  $S \rightarrow B \rightarrow X \rightarrow T$  and update the residual network (see Fig. 2.13c).

Next, we update node potentials (see Fig. 2.14a) and reduce the edge-weights (see Fig. 2.14b). Then we augment the shortest path  $S \rightarrow C \rightarrow X \rightarrow T$  and update the residual network (see Fig. 2.14c). After two more iterations (see Figs. 2.15 and 2.16), the Successive Shortest Path algorithm produces the optimal solution for the Bipartite Weighted Matching problem (i.e., nodes A and B are assigned to node X and nodes C and D are assigned to node Y). It is well known that the Bipartite Weighted Matching problem is a special case of the minimum cost network flow problem [2].



**Fig. 2.15** Successive Shortest Path algorithm: 3rd iteration



**Fig. 2.16** Successive Shortest Path algorithm: 4th iteration

## 2.7 Graph Partitioning

Recently, big data processing platforms (e.g., MapReduce, Apache Hadoop, Apache Spark and GraphLab) have become a growing component for large-scale data management due to their potential ability to distribute large datasets across multiple machines and parallelize the query processing [1]. Data decomposition for parallel query processing follows three general rules to process the query efficiently: (1) data should be placed as close as possible to where it will be used, (2) load balancing should be considered to increase the global system performance, and (3) low communication cost should be considered to improve the query response. Graph Partitioning is a critical step to meet these general rules. For example, big data processing platforms can reduce computational cost for data processing when large sized datasets can be divided into smaller ones of about equal size, with minimum interaction as possible between these smaller datasets. Graph Partitioning decomposes large-sized networks into non-overlapping sub-networks which can efficiently apply a divide-and-conquer scheme to spatial network query optimization (e.g., shortest path computation) [6, 15, 21]. Given a spatial network and a number of partitions (i.e.,  $k$ ), the Graph Partitioning problem is to decompose the spatial network into  $k$  non-overlapping

equal-size sub-networks that minimizes the total weight of the edge-cuts between partitions. The problem is NP-hard for the case  $k = 2$ , which is also called the Minimum Bisection problem. In this section, we review the most popular methods to solve the Graph Partitioning problem.

The basic approach for dealing with Graph Partitioning is to construct an initial solution and iteratively improve the solution with a local search [3]. The Spectral method has been widely used to produce a good initial solution [19]. The Spectral method finds an approximate solution to the Graph Partitioning problem by computing the eigenvectors of a Laplacian Matrix and inferring the edge-cuts from these eigenvectors. The Laplacian Matrix of a graph  $G$  is defined as  $L = D - A$ , where  $D$  is the diagonal matrix expressing node degree and  $A$  is the node-node adjacency matrix.

$$L[i, j] = \begin{cases} \text{degree}(i) & \text{if } i = j \\ -1 & \text{if } i \neq j \text{ and there is an edge}(i, j) \\ 0 & \text{otherwise} \end{cases}$$

The general idea of the Spectral method is to partition the network by utilizing the eigenvector  $v_2$  corresponding to the second smallest eigenvalue  $\lambda_2$  of the Laplacian matrix. Consider the example for the Minimum Bisection problem in Fig. 2.17a. Figure 2.17b shows the Laplacian Matrix for the given input graph. The second smallest eigenvalues  $\lambda_2$  is 0.191 and its corresponding eigenvector  $v_2$  is shown in Fig. 2.17c. The main assumption for the Spectral method is that the element of the eigenvector  $v_2$  is binary (i.e. 1 or  $-1$ ) to represent each partition. In general, elements are not binary, but are distributed over a range of real values, as shown in Fig. 2.17c. One of the ways to handle this issue is to find the median value of elements in the eigenvector and use it as the threshold to bisect a set of nodes. In this example, the median value is 0 and nodes with positive value become one partition and nodes with negative value become the other. Since node  $B$  and  $G$  have the median value (i.e., 0), these nodes are arbitrarily allocated to one of the two partitions. Figure 2.17d shows the output of the Spectral method.

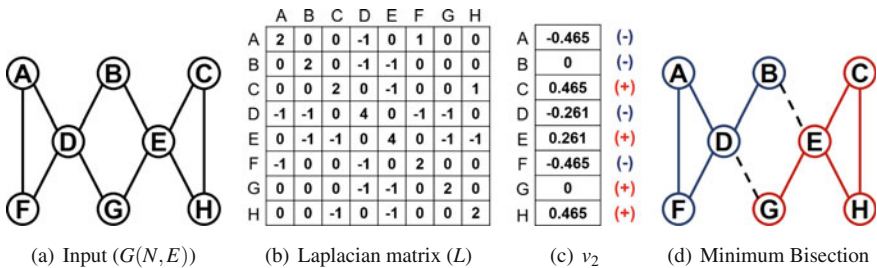


Fig. 2.17 Spectral graph partitioning method

The Kernighan-Lin (KL) and Fiduccia-Mattheyses (FM) methods have been widely used to iteratively improve the solution quality by local search optimization [9, 17]. These methods exchange pairs of nodes, each node from a different partition, and reduce the edge-cuts between partitions. If the network size is too large, Multilevel Partitioning techniques are used to reduce the size of the input network [16]. The Multilevel approach consists of three main phases: (1) coarsening, (2) initial partitioning, and (3) uncoarsening. In the coarsening phase, it iteratively identifies matchings and contracts the edges to reduce the size of the input. For example, Graph  $G_{i+1}$  is constructed from  $G_i$  by finding a maximal matching of  $G_i$  and aggregating the nodes that are incident on each edge of the matching. After that, it constructs an initial partition using spectral or KL/FM methods. In the uncoarsening phase, the partition is projected back to the original graph by progressively disaggregating the nodes. Local refinement can be applied in each step of uncoarsening to improve the solution quality.

## References

1. Agneeswaran VS (2014) Big data analytics beyond hadoop: real-time applications with storm, spark, and more hadoop alternatives. FT Press, Upper Saddle River
2. Ahuja R, et al (1993) Network flows: theory, algorithms, and applications. Prentice Hall, Upper Saddle River
3. Alpert C, Kahng A (1995) Recent directions in netlist partitioning: a survey. *Integr VLSI J* 19(1–2):1–81
4. Bellman R (1958) On a routing problem. *Q Appl Math* 16:87–90
5. Cormen TH (2009) Introduction to algorithms. MIT Press, Cambridge
6. Demetrescu C, Goldberg AV, Johnson DS (2009) The shortest path problem: ninth DIMACS implementation challenge, vol 74. American Mathematical Society, Providence
7. Diestel R (2005) Graph Theory, (4th edn). Graduate texts in mathematics, vol 173. Springer, Heidelberg
8. Dijkstra EW (1959) A note on two problems in connexion with graphs. *Numerische mathematik* 1(1):269–271
9. Fiduccia C, Mattheyses R (1982) A linear-time heuristic for improving network partitions. In: 19th conference on design automation. IEEE, pp 175–181
10. Floyd RW (1962) Algorithm 97: shortest path. *Commun ACM* 5(6):345
11. Ford LR Jr (1956) Network flow theory. Technical report, DTIC Document
12. Garey MR, Johnson DS (2002) Computers and intractability, vol 29. WH Freeman, New York
13. Hart PE, Nilsson NJ, Raphael B (1968) A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans Syst Sci Cybern* 4(2):100–107
14. Johnson DB (1977) Efficient algorithms for shortest paths in sparse networks. *J ACM (JACM)* 24(1):1–13
15. Jung S, Pramanik S (2002) An efficient path computation model for hierarchically structured topographical road maps. *IEEE Trans Knowl Data Eng* 14(5):1029–1046
16. Karypis G et al (1998) A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J Sci Comput* 20(1):359–392
17. Kernighan B, Lin S (1970) An efficient heuristic procedure for partitioning graphs. *Bell Syst Tech J* 49(2):291–307
18. Moore EF (1959) The shortest path through a maze. Bell Telephone System, New York
19. Nascimento MC, De Carvalho AC (2011) Spectral methods for graph clustering—a survey. *Eur J Oper Res* 211(2):221–231

20. Schrijver A (2002) Combinatorial optimization: polyhedra and efficiency, vol 24. Springer Science & Business Media, Heidelberg
21. Shekhar S, Fetterer A, Goyal B (1997) Materialization trade-offs in hierarchical shortest path algorithms. In: International symposium on spatial databases. Springer, pp 94–111
22. Tarjan R (1971) Depth-first search and linear graph algorithms. In: 12th annual symposium on switching and automata theory, pp 114–121. doi:[10.1109/SWAT.1971.10](https://doi.org/10.1109/SWAT.1971.10)
23. Warshall S (1962) A theorem on boolean matrices. J ACM (JACM) 9(1):11–12



<http://www.springer.com/978-3-319-56656-6>

Spatial Network Big Databases

Queries and Storage Methods

Yang, K.; Shekhar, S.

2017, XI, 101 p. 62 illus., Hardcover

ISBN: 978-3-319-56656-6