Homework 5 Solutions

December 2, 2019

3.8

Pouring water. We have three containers whose sizes are 10 pints, 7 pints, and 4 pints, respectively. The 7-pint and 4-pint containers start out full of water, but the 10-pint container is initially empty. We are allowed one type of operation: pouring the contents of one container into another, stopping only when the source container is empty or the destination container is full. We want to know if there is a sequence of pourings that leaves exactly 2 pints in the 7- or 4-pint container.

(a) Model this as a graph problem: give a precise definition of the graph involved and state the specific question about this graph that needs to be answered.

(b) What algorithm should be applied to solve the problem?

(c) Find the answer by applying the algorithm.

Solution

(a) We define a directed graph $G = (V, E)$ as follows:

Node set V: We define a node by a tuple of three integers: (a, b, c), where $0 \leq a \leq 10$, $0 \leq b \leq 7$, $0 \leq c \leq 4$.

The graph thus contains $11 \times 8 \times 5 = 440$ nodes.

Edge set E: We define an edge by a valid pouring from the first to the second node. There are six types of edges:

1. $((a, b, c), (a - \Delta, b + \Delta, c))$: pouring Δ pints of water from container a to b, where
$$\Delta = \min\{a, 7 - b\} \tag{1}$$

2. $((a, b, c), (a - \Delta, b, c + \Delta))$: pouring Δ pints of water from container a to c, where
$$\Delta = \min\{a, 4 - c\} \tag{2}$$

3. $((a, b, c), (a + \Delta, b - \Delta, c))$: pouring Δ pints of water from container b to a, where
$$\Delta = \min\{b, 10 - a\} \tag{3}$$

4. $((a, b, c), (a, b - \Delta, c + \Delta))$: pouring Δ pints of water from container b to c, where
$$\Delta = \min\{b, 4 - c\} \tag{4}$$

5. $((a, b, c), (a + \Delta, b, c - \Delta))$: pouring Δ pints of water from container c to a, where
$$\Delta = \min\{c, 10 - a\} \tag{5}$$

6. $((a, b, c), (a, b + \Delta, c - \Delta))$: pouring Δ pints of water from container c to b, where
$$\Delta = \min\{c, 7 - b\} \tag{6}$$

The graph must be directed because the reverse of an edge is not necessarily a valid pouring.

(b) We can apply BFS to find a shortest pouring sequence starting from the initial state to a goal state. Using DFS, one can also find a valid pouring sequence, but it is not necessarily the shortest.

The full graph does not need to be built up front. It can be constructed on the fly as BFS or DFS is running.

(c) We start the BFS or DFS from the initial state represented by node $(0, 7, 4)$.

We stop if the BFS or DFS reaches any node in the form of $(*, 2, *)$ or $(*, *, 2)$, or if the BFS or DFS returns without find such goal state nodes.

One pouring sequence found by DFS is

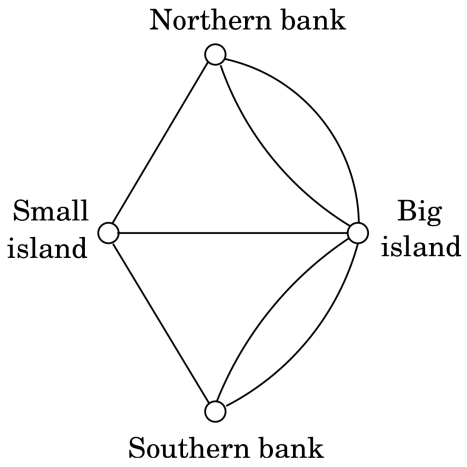
$$(0, 7, 4) \to (4, 7, 0) \to (10, 1, 0) \to (6, 1, 4) \to (6, 5, 0) \to (2, 5, 4) \to (2, 7, 2)$$

3.26

An Eulerian tour in an undirected graph is a cycle that is allowed to pass through each vertex multiple times, but must use each edge exactly once.

This simple concept was used by Euler in 1736 to solve the famous Konigsberg bridge problem, which launched the field of graph theory. The city of Konigsberg (now called Kaliningrad, in western Russia) is the meeting point of two rivers with a small island in the middle. There are seven bridges across the rivers, and a popular recreational question of the time was to determine whether it is possible to perform a tour in which each bridge is crossed exactly once.

Euler formulated the relevant information as a graph with four nodes (denoting land masses) and seven edges (denoting bridges), as shown here.



Notice an unusual feature of this problem: multiple edges between certain pairs of nodes.

(a) Show that an undirected graph has an Eulerian tour if and only if all its vertices have even degree. Conclude that there is no Eulerian tour of the Konigsberg bridges.

(b) An Eulerian path is a path which uses each edge exactly once. Can you give a similar if-and-only-if characterization of which undirected graphs have Eulerian paths?

(c) Can you give an analog of part (a) for directed graphs?

Solution

(a) **Only if (Eulerian tour $\rightarrow$ even degrees):**

Proof. If an Eulerian tour exists in graph G, for any arbitrary vertex $v \in G$, each arrival edge must be paired with a departure edge to make it possible to visit each edge exact once in a cycle. So, the degree of each vertex must be even. □

If (even degrees $\rightarrow$ Eulerian tour):

Proof. (By induction)

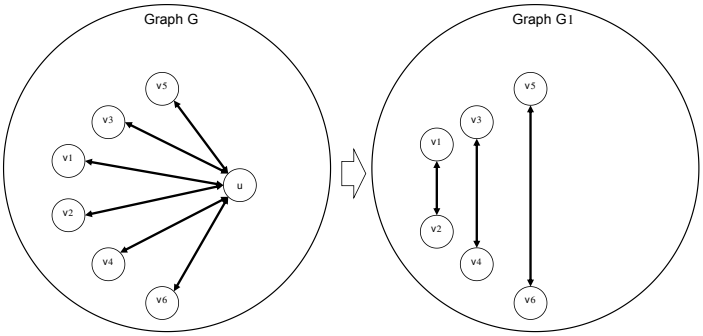
Induction hypothesis: Degrees of all vertices $v \in G$ being even implies the existence of an Eulerian tour.

Base case: graph G has two vertices $|V| = 2$, both with even degrees. There must be an even number of edges between the two nodes. We can find a Eulerian tour by starting from one vertex, choosing an untraversed edge, reaching the other vertex, until all edges are visited.

Induction step: Eulerian path exists for any graph G_0 with $|V| = n \geq 2$ when degrees of vertices are even.

For any graph G has $n + 1 > 2$ vertices with even degrees, pick one vertex $u \in G$. u is connected to vertices v_1, v_2, $\ldots v_{2i-1}, v_{2i}$ by an even number of edges $u \leftrightarrow v_1, u \leftrightarrow v_2, \ldots, u \leftrightarrow v_{2i}$ (duplicate vertices are allowed in $v_1, v_2, \ldots v_{2i}$).

Based on G, create a new graph G_1 with n vertices by removing u as well as all edges connect to u and adding edges $v_1 \leftrightarrow v_2, v_3 \leftrightarrow v_4, \ldots, v_{2i-1} \leftrightarrow v_{2i}$. Since degrees of of vertices of G_1 are also even, based on hypothesis, Eulerian path exists in G_1.



By comparison G_1 and G, we found all paths such as $v_{2i-1} \leftrightarrow v_{2i}$ in G_1 are equivalent to edges $v_{2i-1} \leftrightarrow u \leftrightarrow v_{2i}$ in G, which means Eulerian path also exists in G.

Now, we proved this statement: Known degrees of all vertices are even in both graph G and G_0, if Eulerian path exists in graph G_0 with $|V| = n \geq 2$, then Eulerian path also exists in graph G with $|V| = n + 1 > 2$.

Combined with base case, if direction is proved. □

(b) To have an Eulerian path exactly two vertices (start and end points) must have odd degrees and the rest must have even degrees.

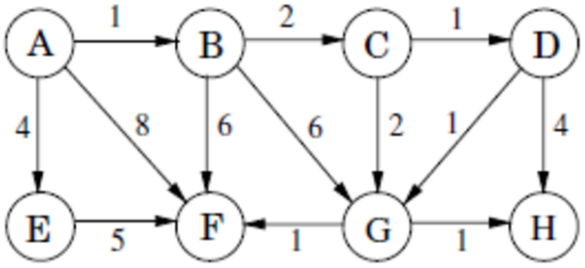
(c) In a directed graph, an Eulerian tour exists if and only if the number of incoming edges are equal to the number of outgoing edges for each vertex.

4.1

Suppose Dijkstra's algorithm is run on the following graph, starting at node A.

(a) Draw a table showing the intermediate distance values of all the nodes at each iteration of the algorithm.

(b) Show the final shortest-path tree.

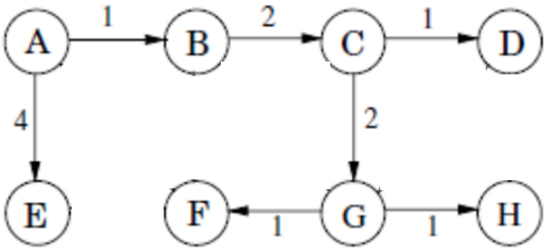


Solution

(a) Table of intermediate distances:

Tree depth	A	B	C	D	E	F	G	H
1	0	1	∞	∞	4	8	∞	∞
2	0	1	3	∞	4	7	7	∞
3	0	1	3	4	4	7	5	∞
4	0	1	3	4	4	6	5	6

(b) The final shortest-path tree:

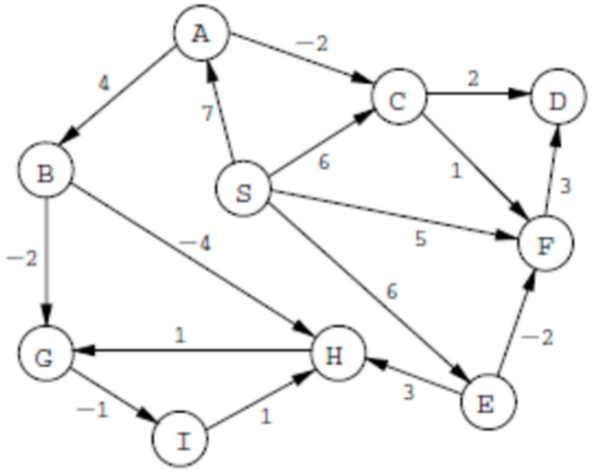


4.2

Just like the previous problem, but this time with the Bellman-Ford algorithm.

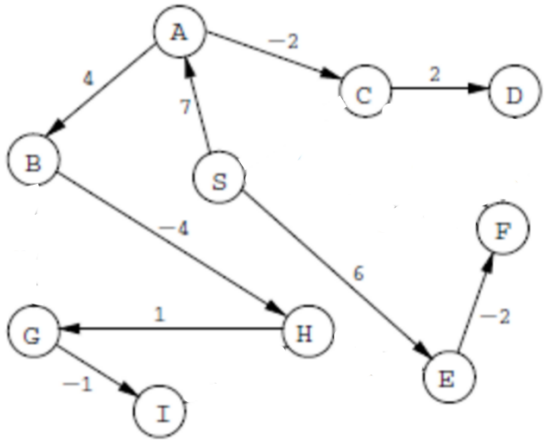
Solution

(a) Table of intermediate distances:



Iteration	S	A	B	C	D	E	F	G	H	I
0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	0	7	∞	6	∞	6	5	∞	∞	∞
2	0	7	11	5	8	6	4	∞	9	∞
3	0	7	11	5	7	6	4	9	7	∞
4	0	7	11	5	7	6	4	8	7	7
5	0	7	11	5	7	6	4	8	7	7
6	0	7	11	5	7	6	4	8	7	7
7	0	7	11	5	7	6	4	8	7	7
8	0	7	11	5	7	6	4	8	7	7
9	0	7	11	5	7	6	4	8	7	7

(b) The final shortest-path tree:



4.3

Squares. Design and analyze an algorithm that takes as input an undirected graph $G = (V, E)$ and determines whether G contains a simple cycle (that is, a cycle which doesnt intersect itself) of length four. Its running time should be at most $O(|V|^3)$.

You may assume that the input graph is represented either as an adjacency matrix or with adjacency lists, whichever makes your algorithm simpler.

Solution 1

In this solution, we examine whether a square can be found based on the BFS traversal of the given graph. It can be found in two scenarios:

1. s → u (dist(u)=1) → v (dist(v)=1) → w (dist(w)=1) → s
2. s → u (dist(u)=1) → v (dist(v)=2) → w (dist(w)=1) → s

---

procedure Find-Simple-Square(G)
Input: An undirected graph $G = (V, E)$
Output: TRUE if G contains a simple square; FALSE otherwise

```
for each node s ∈ V:
    for all u ∈ V:
        dist(u) = ∞
    dist(s)=0
    Q = [s] (queue containing just s)
    while Q is not empty
            u =eject(Q)

            if dist(u) > 2: break the while loop
            else if dist(u)==1:
                 if u has two adjacent nodes v and w such that dist(v)=dist(w)=1
                   return TRUE (a square is detected)
            else if dist(u)==2:
                 if u has one adjacent node w (≠ v=prev(u)) such that dist(w)=1
                   return TRUE (a square is detected)

            for each edge (u, v) ∈ E:
                if dist(v)=∞
                  inject(Q, v)
                  dist(v) = dist(u) + 1
                  prev(v) = u
return FALSE (a square is not detected)
```

Runtime: The outer-for loop has $|V|$ iterations. Within each for-iteration and among all

iterations of the while-loop, each node is visited at most once, each edge at most three times, taking at most $O(|V| + |E|)$ operations. Thus the total runtime is $O(|V|(|V| + |E|))$. It reduces to $O(|V|^3)$ when there is at most one edge between each pair of nodes in the graph.

Solution 2

G has a simple cycle of length four if and only if there exist a pair of vertices x and y that have at least two common neighbors. First, for each vertices $v \in G$, sort adjacency list of v. Each sort takes $O(|V| \log |V|)$ by fast sorting algorithm such as merge sort. For all vertices, it takes $O(|V|^2 \log |V|)$.

Second, for each pair of distinct vertices v and u, obtained adjacency list of v and u, and check the number of comment neighbors between x and y. Since nodes of adjacency list has been sorted, checking common neighbors takes $O(|V|)$ for each pair of x and y by visiting sorted adjacency list of both x and y once. For $O(|V|^2)$ pair of distinct vertices, running time is $O(|V|^3)$

So,the total running time is $O(|V|^3) + O(|V|^2 \log |V|) = O(|V|^3)$.

pseudocode

ADJ(G, v)

1 **return** the adjacency list of node v in graph G

SIMPLESQUARE(G)

```
 1  Input: Graph G = (V, E)
 2  Output: TRUE or FALSE
 3
 4  for each node v ∈ V:
 5        Sort Adj(G, v)
 6
 7  Result = FALSE
 8  for each node v ∈ V:
 9        for each node u ∈ V:
10              adjacent nodes vadj = Adj(G, v)
11              adjacent nodes uadj = Adj(G, u)
12              get intersected nodes of vadj and uadj: inter
13              if (|inter| ≥ 2) Result = TRUE
14  Return Result
```

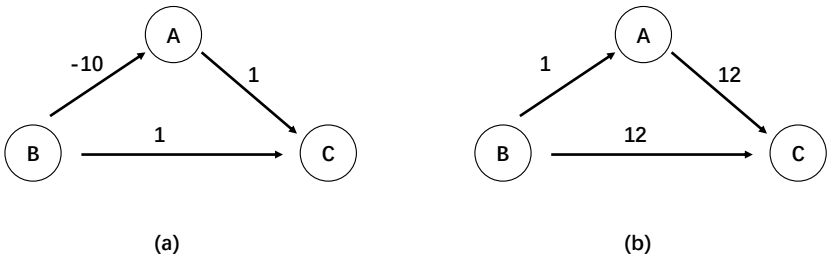
4.8

Professor F. Lake suggests the following algorithm for finding the shortest path from node s to node t in a directed graph with some negative edges: add a large constant to each edge weight so that all the weights become positive, then run Dijkstra's algorithm starting at node s, and return the shortest path found to node t.

Is this a valid method? Either prove that it works correctly, or give a counterexample.

Solution

Professor Lake was wrong. Here is a counter example: In Figure (a), the shortest path from



B to C is $B \to A \to C$. By adding 11 to all edges, we got Figure (b). Now, the shortest path from B to C by Dijkstra's algorithm will be $B \to C$, which does not correspond to the correct shortest path from B to C in the original graph.