Algorithms 2024: Basic Graph Algorithms

(Based on [Manber 1989])

Yih-Kuen Tsay

December 11, 2024

1 Introduction

The Königsberg Bridges Problem

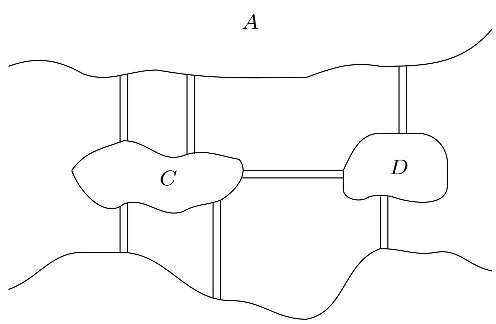


Figure: The Königsberg bridges problem.
Source: redrawn from [Manber 1989, Figure 7.1].

Can one start from one of the lands, cross every bridge exactly once, and return to the origin?

The Königsberg Bridges Problem (cont.)

An abstract model is more convenient to work with:

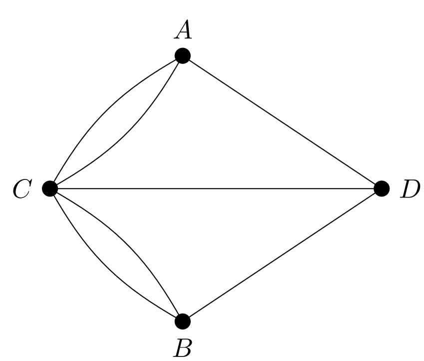


Figure: The graph corresponding to the Königsberg bridges problem.
Source: redrawn from [Manber 1989, Figure 7.2].

Graphs

- A graph consists of a set of vertices (or nodes) and a set of edges (or links, each normally connecting two vertices).
- A graph is commonly denoted as $G(V, E)$, where
  - G is the name of the graph,
  - V is the set of vertices, and
  - E is the set of edges.

Note: we assume that you have learned from a course on Data Structures the basics of graph theory and the representation of a graph by an adjacency matrix or incidence list.

Graphs (cont.)

- Undirected vs. Directed Graph

  /* In a directed graph, every edge (connecting two vertices) is oriented, directing from the start vertex towards the end vertex. */

- Simple Graph vs. Multigraph

  /* In a multigraph, multiple edges are allowed between a pair of vertices; the edges are not labeled (and thus cannot be distinguished). */

- Path, Simple Path, Trail

  /* Path often really means simple path, also called open path, which is a sequence of vertices where each pair of consecutive vertices are connected by an edge. All the vertices on a simple path are distinct, except perhaps the first (start) and the last (end). Trail is just another name for path, but strongly suggests that it may contain a cycle. */

- Cycle, Simple Cycle, Circuit

  /* These are special kinds of paths/trails (also called closed paths/trails) where the start and the end vertices are the same. */

- Degree, In-Degree, Out-Degree

  /* The *degree* of a vertex is the number of edges with the vertex as an end. For a directed graph where the edges are oriented, the *in-degree* of a vertex is the number of edges ending at the vertex, while the *out-degree* of a vertex is the number of edges starting from the vertex. */

- Connected Graph, Connected Components

  /* In a connected graph, every vertex can reach every other vertex via a path. The connected components of a graph are the maximal subgraphs of the entire graph that are by themselves connected. */

- Tree, Forest

  /* Trees are connected graphs without any cycle. A forest is a graph consisting of separate trees (as subgraphs). */

- Subgraph, Induced Subgraph

  /* A vertex-induced subgraph must include every edge in the original graph that connects a pair of the selected vertices. */

- Spanning Tree, Spanning Forest

  /* A *spanning tree* of a graph is a subgraph of the graph that includes all the vertices and is a tree. */

- Weighted Graph

  /* A *weighted graph* is a graph where every edge is associated with a number, usually called its weight, representing its weight, length, or capacity. */

Modeling with Graphs

- Reachability
  - Finding program errors

    /* A program state corresponds to a vertex and there is a directed edge from one vertex to another if the program represented by the first vertex may (in one execution step) become the program state represented by the second vertex. */
  - Solving sliding tile puzzles

    /* A configuration of the sliding tiles corresponds to a vertex and there is a directed edge from one vertex to another if the configuration represented by the first vertex may (in one sliding step) become the configuration represented by the second vertex. */
- Shortest Paths
  - Finding the fastest route to a place
  - Routing messages in networks
- Graph Coloring
  - Coloring maps
  - Scheduling classes

    /* A class corresponds to a vertex and there is an undirected edge between two vertices if the two classes represented by the two vertices are taught by the same instructor. The colors represent the time slots.

    Another interpretation: there is an undirected edge between two vertices if there is a time conflict between the two classes represented by the two vertices. The colors represent the classrooms. */

Eulerian Graphs

Problem 1. *Given an undirected connected graph $G = (V, E)$ such that all the vertices have even degrees, find a circuit P such that each edge of E appears in P exactly once.*

The circuit P in the problem statement is called an *Eulerian circuit*.

Theorem 2. *An undirected connected graph has an Eulerian circuit if and only if all of its vertices have even degrees.*

/* Proof sketch:

(The "only if" part) Suppose the graph has an Eulerian circuit. Each time the circuit enters a vertex, it must also leave the vertex from a different edge. For the first vertex in the circuit, it is left first and entered at last via a different edge. So, every vertex must have an even degree.

(The "if" part) The proof is by induction on the number of edges. Note that the graph must contain at least a simple cycle, as every vertex is of an even degree.

Base case: the graph is a simple cycle (with one edge or more). The cycle clearly is an Eulerian circuit.

Inductive step: Remove a simple cycle from the graph. The remaining part of the graph may consist of several separated components. Each component is connected and every vertex in the component also has an even degree. The induction hypothesis applies to each component. Connecting the removed cycle and the Eulerian circuit of each component, we have an Eulerian circuit for the entire graph. */

2 Depth-First Search

Depth-First Search

/* Depth-first search (DFS) is a systematic way of exhaustively exploring/traversing the vertices (and the edges) of a graph. Given a start vertex, a DFS traversal always chooses to first visit some neighboring vertex of the current vertex until it cannot find any unvisited vertex. If the graph is not connected, several DFS traversals will be needed to exhaust the graph. */

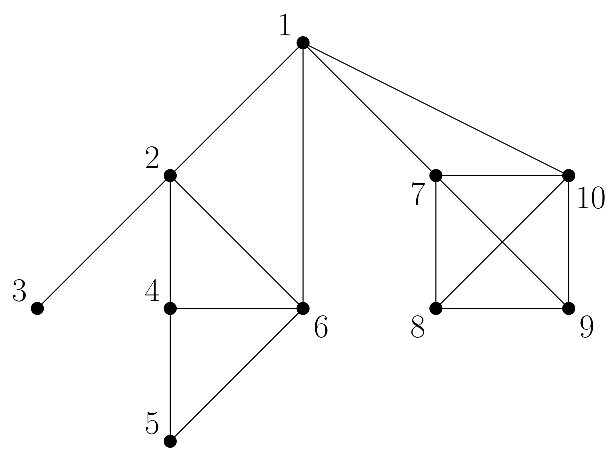


Figure: A DFS for an undirected graph.

Source: redrawn from [Manber 1989, Figure 7.4].

/* The numbers (from 1 through 10) show the order in which the vertices are visited by a particular DFS. */

Depth-First Search (cont.)

Algorithm Depth_First_Search(G, v);
begin
  mark v;
  perform preWORK on v;
  for all edges (v, w) **do**
    if w is unmarked **then**
      $Depth\_First\_Search(G, w)$;
    perform postWORK for (v, w)
end

Depth-First Search (cont.)

Algorithm Refined_DFS(G, v);
begin

mark v;
perform preWORK on v;
for all edges (v, w) **do**
  if w is unmarked **then**
    $Refined\_DFS(G, w)$;
  perform postWORK for (v, w);
perform postWORK_II on v
end

A "Metaphor" of DFS

> Space: the final frontier. These are the voyages of the starship Enterprise. Its five-year mission: to explore strange new worlds. To seek out new life and new civilizations. To boldly go where no man/one has gone before!
>
> – Captain James T. Kirk, *Star Trek*

Connected Components

Algorithm Connected_Components(G);
begin
  $Component\_Number := 1$;
  while there is an unmarked vertex v **do**
    $Depth\_First\_Search(G, v)$
    (preWORK:
      $v.Component := Component\_Number$);
    $Component\_Number := Component\_Number + 1$
end

Time complexity: $O(|E| + |V|)$.

/* Each edge of the input graph is checked twice (once from each end). The algorithm also has to scan possibly many isolated vertices (in some cases, $|V|$ may be larger than $|E|$). */

DFS Numbers

Algorithm DFS_Numbering(G, v);
begin
  $DFS\_Number := 1$;
  $Depth\_First\_Search(G, v)$
  (preWORK:
    $v.DFS := DFS\_Number$;
    $DFS\_Number := DFS\_Number + 1$)
end

Time complexity: $O(|E|)$ (assuming the input graph is connected).

The DFS Tree

Algorithm Build_DFS_Tree(G, v);
begin
  Depth_First_Search(G, v)
  (postWORK:
    if w was unmarked **then**
      add the edge (v, w) to T);
end

The DFS Tree (cont.)

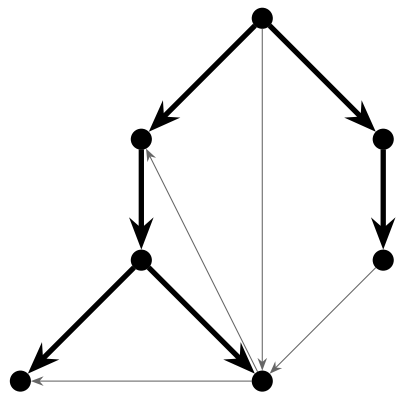


Figure: A DFS tree for a directed graph.
Source: redrawn from [Manber 1989, Figure 7.9].

/* For a directed graph, if all the other vertices may be reached from the starting vertex, then one invocation of the DFS procedure suffices. Otherwise, we can only hope for a spanning forest, not a single spanning tree; and in this case, a DFS needs to be invoked for each vertex that remains unmarked after the previous DFS. */

The DFS Tree (cont.)

Lemma 3 (7.2)**.** *For an undirected graph $G = (V, E)$, every edge $e \in E$ either belongs to the DFS tree T, or connects two vertices of G, one of which is the ancestor of the other in T.*

For undirected graphs, DFS avoids cross edges (that connect vertices on different subtrees of the DFS tree).

Lemma 4 (7.3)**.** *For a directed graph $G = (V, E)$, if (v, w) is an edge in E such that $v.DFS\_Number < w.DFS\_Number$, then w is a descendant of v in the DFS tree T.*

For directed graphs, cross edges must go "from right to left".

/* Here we assume that, when there are multiple vertices to choose for the next visit, the leftmost vertex (according to the layout of the graph) is always chosen first. */

Directed Cycles

Problem 5. *Given a directed graph $G = (V, E)$, determine whether it contains a (directed) cycle.*

Lemma 6 (7.4)**.** *G contains a directed cycle if and only if G contains a back edge (relative to a DFS tree).*

A directed edge that goes from a vertex to one of its ancestor vertices (relative to a DFS tree) is called a *back edge*.

Directed Cycles (cont.)

```
Algorithm Find_a_Cycle(G);
begin
    while there is an unmarked vertex v do
        Depth_First_Search(G, v)
        (preWORK:
            v.on_the_path := true;
         postWORK:
            if w.on_the_path then
                Find_a_Cycle := true;
                halt;
            if w is the last vertex on v's list then
                v.on_the_path := false;)
end
```

Directed Cycles (cont.)

```
Algorithm Refined_Find_a_Cycle(G);
begin
    while there is an unmarked vertex v do
        Refined_DFS(G, v)
        (preWORK:
            v.on_the_path := true;
         postWORK:
            if w.on_the_path then
                Refined_Find_a_Cycle := true;
                halt;
         postWORK_II:
            v.on_the_path := false)
end
```

3 Breadth-First Search

4 Breadth-First Search

Breadth-First Search

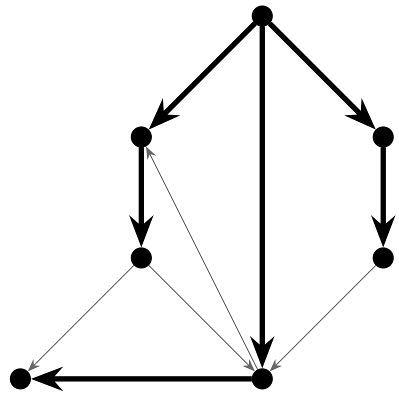


Figure: A BFS tree for a directed graph.

Source: redrawn from [Manber 1989, Figure 7.12].

/* For a directed graph, if all the other vertices may be reached from the starting vertex, then one invocation of the BFS procedure suffices, like in DFS. Otherwise, we can only hope for a spanning forest, not a single spanning tree; and in this case, a BFS needs to be invoked for each vertex that remains unmarked after the previous BFS. */

Breadth-First Search (cont.)

Algorithm Breadth_First_Search(G, v);
begin
  mark v;
  put v in a queue;
  while the queue is not empty **do**
    remove vertex w from the queue;
    perform preWORK on w;
    for all edges (w, x) with x unmarked **do**
      mark x;
      add (w, x) to the BFS tree T;
      put x in the queue
end

Breadth-First Search (cont.)

Lemma 7 (7.5)**.** *If an edge (u, w) belongs to a BFS tree such that u is a parent of w, then u has the minimal BFS number among vertices with edges leading to w.*

Lemma 8 (7.6)**.** *For each vertex w, the path from the root to w in T is a shortest path from the root to w in G.*

Lemma 9 (7.7)**.** *If an edge (v, w) in E does not belong to T and w is on a larger level, then the level numbers of w and v differ by at most 1.*

Breadth-First Search (cont.)

Algorithm Simple_BFS(G, v);
begin
  put v in *Queue*;
  while *Queue* is not empty **do**

remove vertex w from *Queue*;
if w is unmarked **then**
mark w;
perform preWORK on w;
for all edges (w, x) with x unmarked **do**
put x in *Queue*
end

Breadth-First Search (cont.)

Algorithm Simple_Nonrecursive_DFS(G, v);
begin
push v to *Stack*;
while *Stack* is not empty **do**
pop vertex w from *Stack*;
if w is unmarked **then**
mark w;
perform preWORK on w;
for all edges (w, x) with x unmarked **do**
push x to *Stack*
end

5 Topological Sorting

Topological Sorting

Problem 10. *Given a directed acyclic graph $G = (V, E)$ with n vertices, label the vertices from 1 to n such that, if v is labeled k, then all vertices that can be reached from v by a directed path are labeled with labels $> k$.*

Lemma 11 (7.8)**.** *A directed acyclic graph always contains a vertex with indegree 0.*

/* The labeling of the vertices with numbers may immediately start with any vertex that has indegree 0. Once a vertex is labeled, it may be removed along with all the edges directed from it (to some other vertices that remain to be labeled), introducing possibly more vertices now with indegree 0 that may be labeled (with larger numbers). */

Topological Sorting (cont.)

Algorithm Topological_Sorting(G);
initialize $v.indegree$ for all vertices; /* by DFS */
$G\_label := 0$;
for $i := 1$ to n **do**
if $v_i.indegree = 0$ **then** put v_i in *Queue*;
repeat
remove vertex v from *Queue*;
$G\_label := G\_label + 1$;
$v.label := G\_label$;
for all edges (v, w) **do**
$w.indegree := w.indegree - 1$;
if $w.indegree = 0$ **then** put w in *Queue*
until *Queue* is empty

6 Shortest Paths

Single-Source Shortest Paths

Problem 12. *Given a directed graph $G = (V, E)$ and a vertex v, find shortest paths from v to all other vertices of G.*

Shorted Paths: The Acyclic Case

/* When the given graph is acyclic, a path from the source vertex v to some other vertex u can never go via a vertex with a larger label than that of u, according to some topological ordering. So, the shortest paths from the source vertex v to other vertices may be determined one by one, starting from v, and then the vertex next in the topological ordering, and so on till the last vertex (labeled n). Every vertex that is not reachable from the source vertex receives (in the initialization) an ∞ as the length of its shortest path from the source. */

```
Algorithm Acyclic_Shortest_Paths(G, v, n);
{Initially, w.SP = ∞, for every node w.}
{A topological sort has been performed on G, ...}
begin
    let z be the vertex labeled n;
    if z ≠ v then
        Acyclic_Shortest_Paths(G − z, v, n − 1);
        for all w such that (w, z) ∈ E do
            if w.SP + length(w, z) < z.SP then
                z.SP := w.SP + length(w, z)
    else v.SP := 0
end
```

The Acyclic Case (cont.)

```
Algorithm Imp_Acyclic_Shortest_Paths(G, v);
   for all vertices w do w.SP := ∞;
   initialize v.indegree for all vertices;
   for i := 1 to n do
      if v_i.indegree = 0 then put v_i in Queue;
   v.SP := 0;
   repeat
      remove vertex w from Queue;
      for all edges (w, z) do
         if w.SP + length(w, z) < z.SP then
            z.SP := w.SP + length(w, z);
         z.indegree := z.indegree − 1;
         if z.indegree = 0 then put z in Queue
   until Queue is empty
```

Shortest Paths: The General Case

```
Algorithm Single_Source_Shortest_Paths(G, v);
// Dijkstra's algorithm
begin
    for all vertices w do
```

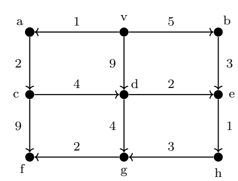
$w.mark := false$;
$w.SP := \infty$;
$v.SP := 0$;
while there exists an unmarked vertex **do**
let w be an unmarked vertex s.t. $w.SP$ is minimal;
$w.mark := true$;
for all edges (w, z) such that z is unmarked **do**
if $w.SP + length(w, z) < z.SP$ **then**
$z.SP := w.SP + length(w, z)$
end

Time complexity: $O((|E| + |V|) \log |V|)$ (using a min heap).

/* Dijkstra's algorithm assumes that the weight of every edge is non-negative. Given the assumption, going from the source vertex v to some other vertex w will never need to pass any other vertex farther than w. So, the algorithm determines one at a time (the lengths of) the paths to the 1-st, 2-nd, $\cdots$, n-th closest vertices (including the source vertex) from the source vertex.

The main loop requires $O(|V|)$ iterations. In each iteration, there is a delete operation on the heap, which takes $O(\log |V|)$ time. The for loop incurs a total of $O(|E|)$ updates to the heap, each taking $O(\log |V|)$ time. */

The General Case (cont.)



	v	a	b	c	d	e	f	g	h
a	0	1	5	∞	9	∞	∞	∞	∞
c	0	①	5	3	9	∞	∞	∞	∞
b	0	①	5	③	7	∞	12	∞	∞
d	0	①	⑤	③	7	8	12	∞	∞
e	0	①	⑤	③	⑦	8	12	11	∞
h	0	①	⑤	③	⑦	⑧	12	11	9
g	0	①	⑤	③	⑦	⑧	12	11	⑨
f	0	①	⑤	③	⑦	⑧	12	⑪	⑨

Figure: An example of the single-source shortest-paths algorithm.

Source: redrawn from [Manber 1989, Figure 7.18].

7 Minimum-Weight Spanning Trees

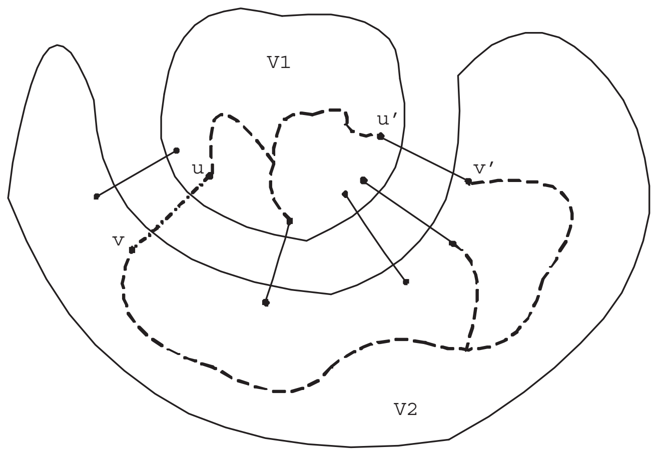
Minimum-Weight Spanning Trees

Problem 13. *Given an undirected connected weighted graph $G = (V, E)$, find a spanning tree T of G of minimum weight.*

Theorem 14. *Let V_1 and V_2 be a partition of V and $E(V_1, V_2)$ be the set of edges connecting nodes in V_1 to nodes in V_2. The edge with the minimum weight in $E(V_1, V_2)$ must be in the minimum-cost spanning tree of G.*

/* Apply the theorem, starting with V_1 containing an arbitrary vertex, and select the minimum-cost edge connecting V_1 to the rest of the graph. Include the connected vertex and we have an expanded V_1 with two vertices. Repeat this process until we cover all vertices of the graph. This is the essence of Prim's algorithm (pseudocode to be given later). */

Minimum-Weight Spanning Trees (cont.)



If $cost(u, v)$ is the smallest among $E(V_1, V_2)$, then $\{u, v\}$ must be in the minimum spanning tree.

/* Suppose $\{u, v\}$ is not chosen. Adding $\{u, v\}$ to the claimed minimum spanning tree will result in a cycle. On the cycle, there must be a heavier $\{u', v'\}$ from $E(V_1, V_2)$. Removing $\{u', v'\}$ would produce a better spanning tree, a contradiction. */

Minimum-Weight Spanning Trees (cont.)

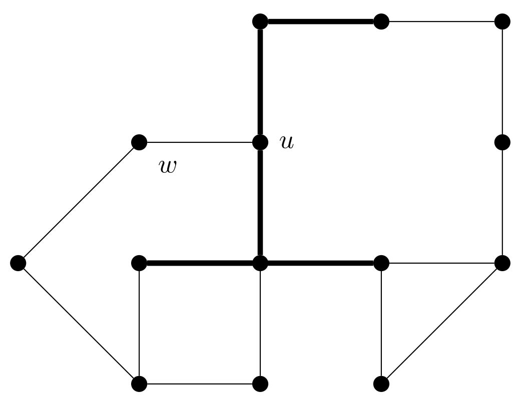


Figure: Finding the next edge of the MCST.

Source: redrawn from [Manber 1989, Figure 7.19].

Minimum-Weight Spanning Trees (cont.)

Algorithm MST(G);
// A variant of Prim's algorithm
begin

```
    initially T is the empty set;
    for all vertices w do
        w.mark := false;  w.cost := ∞;
    let (x, y) be a minimum cost edge in G;
    x.mark := true;
    for all edges (x, z) do
        z.edge := (x, z);  z.cost := cost(x, z);
```

Minimum-Weight Spanning Trees (cont.)

```
    while there exists an unmarked vertex do
        let w be an unmarked vertex with minimal w.cost;
        if w.cost = ∞ then
            print "G is not connected";  halt
        else
            w.mark := true;
            add w.edge to T;
            for all edges (w, z) do
                if not z.mark then
                    if cost(w, z) < z.cost then
                        z.edge := (w, z);  z.cost := cost(w, z)
end
```

Minimum-Weight Spanning Trees (cont.)

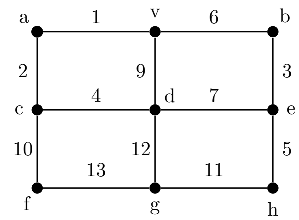
```
Algorithm Another_MST(G);
// Prim's algorithm
begin
    initially T is the empty set;
    for all vertices w do
        w.mark := false;  w.cost := ∞;
    x.mark := true; /* x is an arbitrary vertex */
    for all edges (x, z) do
        z.edge := (x, z);  z.cost := cost(x, z);
```

Minimum-Weight Spanning Trees (cont.)

```
    while there exists an unmarked vertex do
        let w be an unmarked vertex with minimal w.cost;
        if w.cost = ∞ then
            print "G is not connected";  halt
        else
            w.mark := true;
            add w.edge to T;
            for all edges (w, z) do
                if not z.mark then
                    if cost(w, z) < z.cost then
                        z.edge := (w, z);
                        z.cost := cost(w, z)
end
```

Time complexity: same as that of Dijkstra's algorithm.

Minimum-Weight Spanning Trees (cont.)



	v	a	b	c	d	e	f	g	h
v	-	v(1)	v(6)	∞	v(9)	∞	∞	∞	∞
a	-	-	v(6)	$a(2)$	v(9)	∞	∞	∞	∞
c	-	-	v(6)	-	c(4)	∞	$c(10)$	∞	∞
d	-	-	v(6)	-	-	d(7)	$c(10)$	d(12)	∞
b	-	-	-	-	-	b(3)	$c(10)$	d(12)	∞
e	-	-	-	-	-	-	$c(10)$	d(12)	e(5)
h	-	-	-	-	-	-	$c(10)$	h(11)	-
f	-	-	-	-	-	-	$-$	h(11)	-
g	-	-	-	-	-	-	$-$	-	-

Figure: An example of the minimum-cost spanning-tree algorithm. Source: redrawn from [Manber 1989, Figure 7.21].

8 All Shortest Paths

All Shortest Paths

Problem 15. *Given a weighted graph $G = (V, E)$ (directed or undirected) with nonnegative weights, find the minimum-length paths between all pairs of vertices.*

Basic ideas (of Floyd's algorithm):

- Introduce the notion of a k-path, where the largest number of the intermediate vertices is k.
- Induct over the sequence of numbers of the vertices.
- The best m-path from u to v is the best $(< m)$-path from u to m combined with the best $(< m)$-path from m to v.

/* Number the n vertices of the given graph from 1 through n. A directed path from node u to node v is a called a k-path if the largest number of the intermediate nodes (excluding u and v) is k; in the special case where the path is simply a directed edge from u to v (so there is no intermediate vertex), it is called a 0-path. We write $(\leq m)$-path to denote a k-path for some k, $0 \leq k \leq m$; similarly for $(< m)$-path. It is clear that the shortest path (if existing) from u to v is some $(\leq n)$-path. And, that $(\leq n)$-path may be found via an induction on the sequence of numbers of the vertices.

We know the 0-paths between all pairs initially, which is the base case. In the inductive step, we determine the best possible m-path from u to v by combining the best possible $(< m)$-path from u to m with the best possible $(< m)$-path from m to v. Since the weight of every edge is nonnegative (this constraint can be loosened), there is no point of repeating m via a cycle in the path. */

Floyd's Algorithm

```
Algorithm All_Pairs_Shortest_Paths(W);
begin
    {initialization}
    for i := 1 to n do
       for j := 1 to n do
          if (i, j) ∈ E then W[i, j] := length(i, j)
          else W[i, j] := ∞;
    for i := 1 to n do W[i, i] := 0;

    for m := 1 to n do  {the induction sequence}
       for x := 1 to n do
          for y := 1 to n do
             if W[x, m] + W[m, y] < W[x, y] then
                W[x, y] := W[x, m] + W[m, y]
end
```

/* The graph may contain edges with negative weights, as long as there is no cycle whose total weight is negative. */

Transitive Closure

Problem 16. *Given a directed graph $G = (V, E)$, find its transitive closure.*

/* The transitive closure of a graph is obtained from the graph by adding edges such that, whenever there is an edge (x, m) from x to m and an edge (m, y) from m to y, there must also be an edge (x, y) from x to y. If vertex v is reachable (via a directed path) from another vertex u in a graph, then there will be an edge (u, v) in the transitive closure of the graph. */

```
Algorithm Transitive_Closure(A);
begin
    {initialization omitted}
    for m := 1 to n do
        for x := 1 to n do
            for y := 1 to n do
                if A[x, m] and A[m, y] then
                    A[x, y] := true
end
```

Transitive Closure (cont.)

```
Algorithm Improved_Transitive_Closure(A);
begin
    {initialization omitted}
    for m := 1 to n do
        for x := 1 to n do
            if A[x, m] then
                for y := 1 to n do
                    if A[m, y] then
                        A[x, y] := true
end
```