Course Notes for CS336: Graph Theory

Jayadev Misra
The University of Texas at Austin

5/11/01

Contents

1 Introduction **1**
1.1 Basics 2
1.2 Elementary theorems 3
1.3 Graph representation: 4

2 Search Algorithms **5**
2.1 Breadth-First search 5
2.2 Depth-First Search 6

3 All Pairs Connectivity **6**
3.1 Warshall's Algorithm 7

4 Shortest Path Algorithm **8**

5 Minimum Spanning Tree **11**
5.1 Kruskal's algorithm 12
5.2 Dijkstra's algorithm for minimum spanning tree 13

1 Introduction

Reading Assignment and Homework From Rosen,
Reading Assignment: 7.1, 7.2 (omit applications in Page 450), 7.3 (omit isomorphism, Multigraphs, Incidence matrix), 7.4, 7.5 Homework:

7.1: 4, 6, 8, 10, 18
7.2: 2, 14, 16, 20, 26
7.3: 8, 24

1.1 Basics

Examples of graphs:
Road network
Prerequisite structure in CS
An electrical circuit

Terms

Vertex/node, edge

directed/undirected

path/cycle; simple path/cycle

path length

degree

special kinds of graphs:

acyclic
completely connected
Bipartite
tree (directed and undirected)

Show that if there is path between a pair of nodes there is a simple path. Similarly for cycles.

An undirected graph is *is connected* if there is a path between every pair of nodes. A directed graph is *strongly connected* if there is a path between every pair of nodes.

Exercise: A directed graph is strongly connected iff for any node x there is a path from x to every other node and a path from every other node to x.

Some algorithmic questions In the following, x and y are nodes in either an undirected or directed graph.

1. Is there a path from x to y?
2. Find all nodes that can reach x. Also, that can be reached from x.
3. Find the connectivity matrix.
4. Given lengths on edges, find the shortest path from x to y. Find shortest paths between all pairs of nodes.
5. Find the minimum spanning tree in an undirected graph.

1.2 Elementary theorems

Theorem: In an undirected graph, number of nodes of odd degree is even.

Proof: Let U be the nodes of odd degree and V of even degree. Then $\sum_{u \in U}(1 + deg(u)) + \sum_{v \in V} deg(v)$ is even since each term is even. $\sum_{u \in U}(1 + deg(u)) + \sum_{v \in V} deg(v) = |U| + \sum_{u \in U} deg(u) + \sum_{v \in V} deg(v)$. The term $\sum_{u \in U} deg(u) + \sum_{v \in V} deg(v)$ is $2\times$ the number of edges in the graph, which is even. So, $|U|$ is even. □

Theorem: A cycle in a bipartite graph is of even length (has even number of edges).

Proof: Nodes in a bipartite graph can be divided into two subsets, L and R, where the edges are all cross-edges, i.e., incident on a node in L and in R. Consider a cycle and label its nodes "L" or "R" depending on which set it comes from. The node labels alternate; therefore, a cycle has an even number of nodes (hence, an even number of edges). □

Exercise: Show that if all cycles in a graph are of even length then the graph is bipartite. As a corollary, a tree is bipartite.

Exercise: Color the edges of a bipartite graph either red or blue such that for each node the number of incident edges of the two colors differ by at most 1.

Euler paths Consider the undirected graph shown in Figure 1. A cycle —not necessarily simple— which includes every edge exactly once is called an *Euler cycle*. Does the following graph have an Euler cycle?

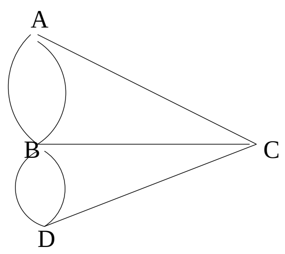

Figure 1: Example graph for Euler path

Theorem: An Euler cycle exists in an undirected graph iff every node has an even degree.

An Euler path is a path which includes every edge exactly once.

Theorem: An Euler path exists in an undirected graph iff exactly two nodes have odd degree.

	A	B	C	D	E
A	0	1	0	0	1
B	0	0	1	0	0
C	1	0	0	0	0
D	1	1	0	0	0
E	0	0	0	1	0

Table 1: Adjacency Matrix M

DeBruijn sequences An application of Euler's theorem is in finding binary sequences which contain all binary strings of length n, for some given n, as substrings, counting wrap-around. For $n = 1$, there are two such sequences, 01 and 10. For $n = 2$ there are 4 binary strings of length 2, and we may expect the required sequence to have 8 bits. However, the following 4-bit sequence works: 0011. For n-bit strings we need at least a 2^n bit sequence. It is possible to construct one, by using Euler's theorem.

1.3 Graph representation:

Consider the graph shown in Figure 2.

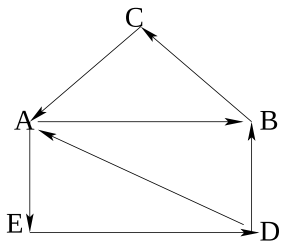

Figure 2: A typical directed graph

This graph can be represented by a matrix M, called the adjacency matrix, as shown below. There is a row and column for each node; $M[i, j] = 0$ if there is no edge from i to j, if there is an edge $M[i, j] = 1$. Note that $M[i, i] = 0$ unless there is a self-loop around i.

linked list representation The graph can also be represented by a set of linked lists, one linked list for each node. The linked list for node A, for instance, lists all the nodes that are the successors of A.

A: B, E
B: C
C: A
D: A, B

E: D

Undirected graph For an undirected graph the adjacency matrix is symmetric, so only half the matrix needs to be kept. The linked list representation has two entries for an edge (u, v), once in the list for u and once for v.

2 Search Algorithms

2.1 Breadth-First search

Given a directed graph find all the nodes reachable from a specific node.

Let r be the node whose successors we wish to mark. Let the *distance* of a node x be the minimum number of edges in a path from r to x. If x is reachable from r then its distance is at most $n-1$, where n is the number of nodes. If x is unreachable then its distance is taken to be ∞. The following algorithm *marks* all the nodes reachable from r in order of their distances.

$i := 0$; $D := \{r\}$; Mark the nodes in D;
while $i < n$ **do**
  $i := i + 1$
  $D :=$ unmarked successors of the nodes in D ;
  Mark the nodes in D;
od

We have the invariant: D is the set of nodes at distance i from r, $0 \le i < n$, and all nodes in D are marked.

There are $O(n)$ iterations. The number of steps is proportional to the number of marked edges and this is bounded by $O(m)$, where m is the number of edges. If we use adjacency list as the representation scheme then the neighbors of each node are easily computed.

Exercise: Show that if node x is at distance k, then x is placed in D when $i = k$.

Exercise: Why are only unmarked successors placed in D?

Exercise: Implement the step

$D :=$ unmarked successors of the nodes in D

Exercise: Modify the algorithm to find the paths from r to every reachable node.

Breadth-First search tree

	a	b	c	d
a	0	0	0	0
b	1	0	1	0
c	1	0	0	1
d	0	0	1	0

Table 2: Adjacency Matrix A

	a	b	c	d
a	0	0	1	0
b	1	0	0	1
c	0	0	1	1
d	1	0	0	1

Table 3: Matrix A^2

2.2 Depth-First Search

3 Transitive Closure

Given the adjacency matrix of a directed graph compute the reachability matrix; in the reachability matrix R, $R[i, j]$ is 1 if there is a non-trivial path (of 1 or more edges) from i to j and $R[i, j]$ is 0 otherwise. Observe that $R[i, i]$ is 1 iff i is on a cycle; if all $R[i, i]$s are 0 then the graph is acyclic.

Consider the graph in Figure 3; we will compute its reachability matrix. Its adjacency matrix A is shown in Table 2.

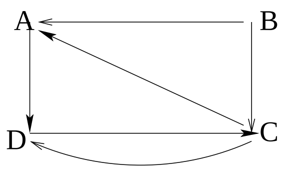

Figure 3: connectivity in a directed graph

Let us compute A^2, i.e., $A \times A$ where we reduce each nonzero entry to 1. This matrix is shown in Table 3.

Note that $A^2[i, j] = 1$ iff there is a path of length 2 (that is having two edges) from i to j. Since there is no 2-edge path from b to c the corresponding entry is 0 (there is a 1-edge path from b to c).

Let us treat the matrix entries as truth values —1 for *true* and 0 for *false*— and define matrix multiplication as follows. We use logical or ($\vee$) in place of $+$ and logical and ($\wedge$) in place of $\times$. Matrix A^0 is the identity matrix I and $A^{t+1} = A \times A^t$.

We claim that that for all t, $t \geq 0$, $A^t[i,j] = 1$ iff there is a path of length t (that is having t edges) from i to j. The proof is by induction on t.
Case $t = 0$**:** We have to show that $A^0[i,j] = 1$ iff there is a path of length 0 from i to j. Since $A^0 = I$, $A^0[i,j] = 1$ iff $i = j$; and there is a path of 0 length from each node to itself.

Case $t + 1, t \geq 0$**:**

$$
\begin{array}{cl}
 & A^{t+1}[i,j] = 1 \\
\equiv & \{\text{definition of matrix multiplication}\} \\
 & \langle \exists u :: A[i,u] = 1 \ \wedge \ A^t[u,j] = 1 \rangle \\
\equiv & \{\text{meaning of } A[i,u] = 1\} \\
 & \langle \exists u :: \text{ there is an edge from } i \text{ to } u \ \wedge \ A^t[u,j] = 1 \rangle \\
\equiv & \{\text{meaning of } A^t[u,j] = 1 \text{ by induction}\} \\
 & \langle \exists u :: \text{ there is an edge from } i \text{ to } u \\
 & \wedge \text{ there is a path of length } t \text{ from } u \text{ to } j \rangle \\
\equiv & \{\text{definition of path}\} \\
 & \text{there is a path of length } t+1 \text{ from } i \text{ to } j
\end{array}
$$

□

The reachability matrix is given by

$$R = A + A^2 + \ldots$$

that is, $R[i,j] = 1$ iff there is some t for which $A^t[i,j] = 1$. It is sufficient to compute the above sum up to A^n, where there are n nodes in the graph. Note that it is necessary to compute up to A^n, because a node may be connected to itself in only a cycle that includes all nodes.

The time complexity of the algorithm is $O(n^4)$, because: (1) computation of A^t requires a matrix multiplication which takes $O(n^3)$, and (2) there are $O(n)$ such matrices to compute.

3.1 Warshall's Algorithm

An entirely different approach, due to Warshall, results in an $O(n^3)$ algorithm. We will compute a sequence of matrices, $W_0, \ldots, W_t, \ldots W_n$, but computation of each matrix will take only $O(n^2)$ steps, resulting in an $O(n^3)$ algorithm.

Let the nodes be labelled $0, \ldots, (n-1)$. Define

$W_t[i,j] =$
there is a path from i to j in which all intermediate nodes are less than t.

Then,

$W_0 = A$, and
$W_n = R$.

We next show how to compute W_{t+1} from W_t, $t \geq 0$.

$W_{t+1}[i,j]$
$=$ $\{$definition of $W_{t+1}[i,j]\}$
there is a path from i to j in which all intermediate nodes are $< t+1$.
$=$ $\{$arithmetic$\}$
there is a path from i to j in which all intermediate nodes are $\leq t$.
$=$ $\{$case analysis$\}$
(there is a path from i to j in which all intermediate nodes are $\leq t$
and t is on the path)
$\vee$
(there is a path from i to j in which all intermediate nodes are $\leq t$
and t is not on the path)
$=$ $\{$simplification$\}$
(there is a path from i to t in which all intermediate nodes are $< t$
$\wedge$ there is a path from t to j in which all intermediate nodes are $< t$)
$\vee$
(there is a path from i to j in which all intermediate nodes are $< t$)
$=$ $\{$Use the definition of $W\}$
$(W_t[i,t] \ \wedge \ W_t[t,j]) \ \vee \ W_t[i,j]$

Exercise For an undirected graph whose edges are weighted, define the *span* of a node to be the maximum distance (i.e., the length of the shortest path) to any node. A node is a *center* if it has the smallest span. Suggest an algorithm for locating a center.

4 Shortest Path Algorithm

Given is a directed graph each edge of which has a positive length. The length of a path is the sum of the edge-lengths along that path. It is required to find the shortest path from a given node, *source*, to another specified node. Given in Figure 4 is an example graph in which the shortest path from A to D is $ABCD$ with length 10 and AF is the shortest path from A to F.

We describe an algorithm, due to Dijkstra, that solves the shortest path problem in $O(n^2)$ steps, where n is the number of nodes. The algorithm finds the shortest path from *source* to all nodes.

Outline of the algorithm In the following, *path* refers to a path from the *source*. For a node x let s_x denote the length of the shortest path to x; $s_{source} = 0$.

The algorithm finds the shortest paths to the various nodes in order of their lengths. Let E denote the set of nodes to which the shortest paths have been found and F, the set of remaining nodes. A step of the algorithm consists of finding a node v in F such that $s_v = (\min\ x : x \in F : s_x)$; then v is moved from F to E. Since the lengths of the shortest paths, s_x for x in F, would not be known, we use a different technique to locate v.

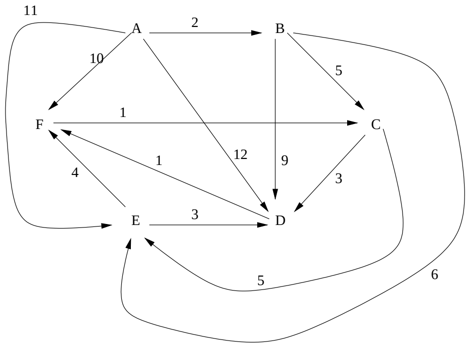

Figure 4: Example graph for shortest path

A path that uses only nodes from E as intermediate nodes will be called a *run*. For any x in F, we let d_x be the length of the shortest run to x. In Lemma 1, we show that the vertex in F that has the shortest run also has the shortest path, i.e., if $d_v = (\min\ x : x \in F : d_x)$ then $s_v = (\min\ x : x \in F : s_x)$, and further $s_v = d_v$. Therefore, the node v with the minimum d-value in F can be moved to E. In Lemma 2 we show how d_x, for the remaining x in F, can be updated efficiently when v is moved from F to E. The algorithm terminates when F is empty.

Development of the Algorithm We postulate the following invariants.

- (P0) $(\forall x : x \in E : d_x = s_x)$.
- (P1) $(\forall x, y : x \in E,\ y \in F : s_x \leq s_y)$.
- (P2) $(\forall x : x \in F : d_x =$ length of the shortest run to $x)$.

The assignments given below establish the invariants (P0, P1, P2) initially. In the following, V is the set of nodes in the graph.

$$E := \ \phi;\ F := \ V; (\forall x : x \in F \wedge x \neq \mathit{source} : d_x := \infty);\ d_{source} := 0$$

Lemma 1: For v in F suppose $d_v = (\min\ x : x \in F : d_x)$. Then, $s_v = d_v$, and $s_v = (\min\ x : x \in F : s_x)$.

Proof: Let u in F satisfy $s_u = (\min\ x : x \in F : s_x)$. We show $s_u = d_u = s_v = d_v$.

The shortest path to u does not include any node w from F as an intermediate node, because then $s_w < s_u$ (since edge lengths are positive), contradicting the definition of s_u. Hence, the shortest path to u is a run, and it is, by definition, the shortest run. Therefore, $s_u = d_u$.

$$
\begin{array}{ll}
 & s_u \\
= & \{\text{see argument above}\} \\
 & d_u \\
\geq & \{d_v = (\min\ x : x \in F : d_x)\} \\
 & d_v \\
\geq & \{d_v \text{ is a path length to } v;\ s_v \text{ is the length of the shortest path to } v\} \\
 & s_v \\
\geq & \{s_u \text{ is the minimum over all } s_x,\ x \in F\} \\
 & s_u
\end{array}
$$

Hence, $s_u = d_u = d_v = s_v$. □

Lemma 1 guarantees that moving v from F to E preserves the invariants (P0, P1). Next, we show how to recompute d_x, for all $x \in F$, so that invariant (P2) is preserved.

Consider all paths to x in which the intermediate nodes are from $E \cup \{v\}$; d_x is to be set to the length of the shortest such path. Partition these paths into (1) those in which v does not appear, and (2) those in which v appears. The shortest path length in (1) is d_x, from invariant (P2). The shortest path length in (2) is – see lemma 2 – $d_v + l[v, x]$, where $l[v, x]$ is the length of the edge from v to x (it is ∞ if there is no such edge). Hence, d_x is set to $\min(d_x, s_v + l[v, x])$.

Lemma 2: Consider the paths to a node x in F in which (1) the intermediate nodes are from $E \cup \{v\}$, where v is as in Lemma 1, (2) $s_u \leq s_v$ for all u in E, and (3) v appears on the path. The length of the shortest such path is $s_v + l[v, x]$. Proof: Let the shortest path, p, that satisfies conditions (1,2,3) has u as the next node after v. If $u \neq x$ then u is from E, from (1). Replace the initial segment of p up to u by the shortest path to u, thus lowering the total path length by $s_v + l[v, u] - s_u$, a positive amount since $s_u \leq s_v$, from (2), and $l[v, u] > 0$. Therefore, the node following v on p is x, and the length of p is $s_v + l[v, x]$.

The Complete Algorithm

$E :=\ \phi;\ F :=\ V; (\forall x : x \in F \wedge x \neq source : d_x := \infty);\ d_{source} := 0;$
while $F \neq \phi$ **do**
  let v **satisfy** $d_v = (\min\ x : x \in F : d_x)$;
  $E,\ F :=\ E \cup \{v\},\ F - \{v\}$;
  $\langle \forall x : x \in F \wedge x$ neighbor of $v : d_x := \min(d_x, d_v + l[v, x]) \rangle$
od

Performance of the Algorithm Each iteration takes at most $O(n)$ time: (1) the smallest d can be computed in $O(n)$ time and (2) updating d_x for all remaining x in F takes $O(n)$ time. There are n iterations; hence, the algorithm is $O(n^2)$.

5 Minimum Spanning Tree

Reading Assignment and Homework From Rosen,
Reading Assignment: 7.6, 8.6
Homework:
7.6: 2, 4, 14, 16
8.6: 2, 6, 10, 11, 12.

Given is an undirected graph each edge of which has a positive length. A subset of edges is called a *tree* if there is no cycle in this subset. A tree is a spanning tree if it connects all the nodes, i.e., there is a path between any pair of nodes. It is required to find a spanning tree sum of whose edge lengths is the minimum; such a spanning tree is called a *minimum spanning tree*. Henceforth, we assume that the edge lengths are distinct.

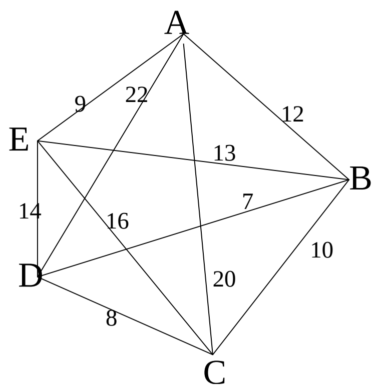

Figure 5: Example graph for minimum spanning tree

A spanning tree for the graph in Figure 5 is $\{BD, CB, AE, CE\}$. This has a edge-weight of 42 whereas $\{BD, CD, AE, AB\}$ has a edge-weight of 36.

Exercise: Is the minimum spanning tree unique? Assume that the edge lengths are distinct.

Properties of Minimum Spanning Tree Let M be a a minimum spanning tree.

- (P1) There are $n - 1$ edges in a spanning tree.
- (P2) There is exactly one path connecting a pair of nodes in a spanning tree.
  Proof: Since the tree is spanning every pair of nodes is connected by at least one path. If there are multiple paths then there is a cycle.

- (P3) Adding an edge to a spanning tree creates a cycle.
  Proof: Consider addition of an edge (x, y). There is a path between (x, y) using only the tree edges, and using the added edge a cycle is completed.

- (P4) Removing any edge from a cycle as in (P3) creates a spanning tree.
  Proof: First, we show that every pair of nodes (x, y) is connected. If both (x, y) are on the created cycle then there is a path between them even after removing an edge from the cycle. If either one of x, y is not on the cycle then there is a path using the original tree edges. Next, we show that there is no cycle after removing an edge from the created cycle. There are now $n - 1$ edges. If each pair of nodes is connected then there is no cycle.

- (P5) Let e be any edge outside the minimum spanning tree. The edges on the cycle created by adding e have lower lengths than that of e.
  Proof: Let C be the cycle created by adding e to the minimum spanning tree. From (P3, P4), any edge f in C can be replaced by e to create a spanning tree. The length of the resulting spanning tree is lower if the length of e is lower than that of f, a contradiction if the original spanning tree is minimum.

5.1 Kruskal's algorithm

The following algorithm, due to Kruskal, finds a minimum spanning tree, T. Initially, T is empty. Scan the edges in increasing order of length; for edge e, if e forms a cycle with T then discard the edge, otherwise, add e to T. The algorithm terminates when T has $n-1$ edges. The algorithm operating on Figure 5 produces the spanning tree that consists of the edges $\{BD, CD, AE, AB\}$; the edge BC was discarded because it forms a cycle with $\{BD, CD\}$.

Theorem: Let the set of scanned edges be E and the minimum spanning tree be M. Then, $T = M \cap E$ is an invariant of the algorithm.
Proof: Initially, the invariant holds because T, E are both empty. Let e be the next edge to be scanned. We show that

$e \notin M \equiv T \cup \{e\}$ has a cycle.

The proof is by mutual implication.

1. $e \notin M \Rightarrow T \cup \{e\}$ has a cycle:

$$
\begin{array}{ll}
 & e \notin M \\
\Rightarrow & \{\text{P5: there is a set of edges } c \text{ in } M \text{ that form a cycle with } e; \\
 & \quad \text{all the edges in } c \text{ have lengths lower than } e\} \\
 & c \cup \{e\} \text{ is a cycle },\ c \subseteq M,\ c \subseteq E \\
\Rightarrow & \{\text{predicate calculus}\} \\
 & c \cup \{e\} \text{ is a cycle },\ c \subseteq M \cap E \\
\Rightarrow & \{\text{invariant: } T = M \cap E\}
\end{array}
$$

$$
\begin{array}{ll}
 & c \cup \{e\} \text{ is a cycle },\ c \subseteq T \\
\Rightarrow & \{\text{Simple graph theory}\} \\
 & T \cup \{e\} \text{ has a cycle}
\end{array}
$$

2. $T \cup \{e\}$ has a cycle $\Rightarrow e \notin M$:

$$
\begin{array}{ll}
 & T \cup \{e\} \text{ has a cycle} \\
\Rightarrow & \{T = M \cap E.\ \text{Hence, } T \subseteq M\} \\
 & M \cup \{e\} \text{ has a cycle} \\
\Rightarrow & \{M \text{ is a spanning tree}\} \\
 & e \notin M
\end{array}
$$

□

Observation: If the number of edges in $T = n - 1$, where n is the number of nodes in the graph, then $T = M$.

Proof: We have the invariant $T = M \cap E$. Hence, $T \subseteq M$. The sizes of M, T are both $n - 1$. Therefore, $T = M$.

Performance of the algorithm A simple analysis shows that Kruskal's algorithm can be implemented in $O(m \times n)$ steps, where m is the number of edges and n the number of nodes. First, sort all the edges by their lengths; this takes $O(m \log m)$ steps, which is no more than $O(m \times n)$. Next, we have to consider each edge in this sequence and determine if it makes a cycle with the edges that have already been chosen. To detect a cycle in a naive way takes about $O(n)$ steps, and we may have to consider all m edges, thus expending $O(m \times n)$ steps. A more sophisticated implementation takes $O(m \log n)$ steps in the worst case; if the graph is dense then $m = O(n^2)$, so the complexity could be as high as $O(n^2 \log n)$.

5.2 Dijkstra's algorithm for minimum spanning tree

Dijkstra's algorithm for minimum spanning tree operates in $O(n^2)$ steps where n is the number of nodes. So for a dense graph —i.e., one in which the number of edges is $O(n^2)$— this algorithm is expected to perform better than Kruskal's. For a sparse graph, Kruskal's algorithm may be superior.

The algorithm operates as follows. The nodes are partitioned into two sets, L and R, where L is always non-empty and T is a subset of the edges. Initially, L consists of one arbitrary node, R has the remaining nodes and T is empty.

As long as $|L| < n$ the following step is executed. Consider the cross edges between L and R, i.e., (x, y), where $x \in L$ and $y \in R$. Among all such edges let (u, v) be the edge of the smallest length. Add (u, v) to T and v to L.

Correctness of the algorithm Let M be the minimum spanning tree of the graph. We show that $T \subseteq M$ is an invariant.

The initialization establishes the invariant because T is empty. Also, T is a spanning tree for the nodes in L; hence, when $|L| = n$, T is a spanning tree for the graph. From the invariant $T = M$.

Next, we show that each step preserves the invariant, i.e., if (u, v) is added to T, then $(u, v) \in M$. Suppose $(u, v) \notin M$; then adding (u, v) to M creates a cycle, from (P3). Label each node in the cycle L or R depending on the set it belongs to. Since $u \in L$ and $v \in R$, there are two adjacent nodes in the cycle labeled L and R. Since it is a cycle there are two other adjacent nodes, say x and y, labeled L and R. From (P5), each edge in the cycle including (x, y) has smaller length than (u, v). This contradicts the choice of (u, v) as the edge of smallest length joining a node in L to a node in R.

Implementation of the algorithm For every node in R we keep its cheapest connection (i.e., edge of the smallest length) to a node in L. There are at most n such edges because R has at most n nodes. For the nodes in R that are not connected to any node in L the value of the cheapest connection is ∞. Initially, these edges consist of all the edges incident on the single node in L.

It takes at most $O(n)$ steps to find the cheapest connection between L and R, by scanning over the individual cheapest connections. Once a node v is moved from R to L the cheapest connections have to be recomputed: for every node y in R, its cheapest connection is compared against (v, y), and the cheaper of the two edges is retained. This takes constant time for each node; hence at most $O(n)$ time for the entire recomputation.

There are $O(n)$ steps because each step adds a single edge to T. Each step takes $O(n)$ time; therefore, the algorithm is $O(n^2)$.