2.2.35. (∗) *Tarry's Algorithm* (as presented by D.G. Hoffman). Consider a castle with finitely many rooms and corridors. Each corridor has two ends; each end has a door into a room. Each room has door(s), each of which leads to a corridor. Each room can be reached from any other by traversing corridors and rooms. Initially, no doors have marks. A robot started in some room will explore the castle using the following rules.

1) After entering a corridor, traverse it and enter the room at the other end.
2) Upon entering a room with all doors unmarked, mark I on the door of entry.
3) In a room with an unmarked door, mark O on such a door and use it.
4) In a room with all doors marked, exit via a door not marked O if one exists.
5) In a room with all doors marked O, stop.

Prove that the robot traverses each corridor exactly twice, once in each direction, and then stops. (Hint: Prove that this holds for the corridors at every reached vertex, and prove that every vertex is reached. Comment: All decisions are completely local; the robot sees nothing other than the current room or corridor. Tarry's Algorithm [1895] and others are described by König [1936, p35–56] and by Fleischner [1983, 1991].)

2.3. Optimization and Trees

"The best spanning tree" may have various meanings. A **weighted graph** is a graph with numerical labels on the edges. When building links to connect locations, the costs of potential links yield a weighted graph. The minimum cost of connecting the system is the minimum total weight of its spanning trees.

Alternatively, the weights may represent distances. In these case we define the length of a path to be the sum of its edge weights. We may seek a spanning tree with small distances. When discussing weighted graphs, **we consider only nonnegative edge weights.**

We also study a problem about finding good trees to encode messages.

MINIMUM SPANNING TREE

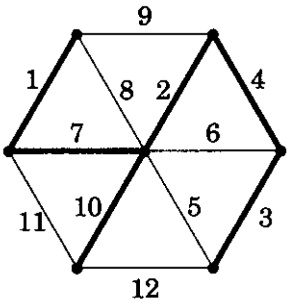
In a connected weighted graph of possible communication links, all spanning trees have $n - 1$ edges; we seek one that minimizes or maximizes the sum of the edge weights. For these problems, the most naive heuristic quickly produces an optimal solution.

2.3.1. Algorithm. (Kruskal's Algorithm - for minimum spanning trees.)
Input: A weighted connected graph.
Idea: Maintain an acyclic spanning subgraph H, enlarging it by edges with low weight to form a spanning tree. Consider edges in nondecreasing order of weight, breaking ties arbitrarily.
Initialization: Set $E(H) = \emptyset$.
Iteration: If the next cheapest edge joins two components of H, then include it; otherwise, discard it. Terminate when H is connected. ■

Theorem 2.3.3 verifies that Kruskal's Algorithm produces an optimal tree. Unsophisticated locally optimal heuristics are called **greedy algorithms**. They usually don't guarantee optimal solutions, but this one does.

In a computer, the weights appear in a matrix, with huge weight on "unavailable" edges. Edges of equal weight may be examined in any order; the resulting trees have the same cost. Kruskal's Algorithm begins with a forest of n isolated vertices. Each selected edge combines two components.

2.3.2. Example. Choices in Kruskal's Algorithm depend only on the order of the weights, not on their values. In the graph below we have used positive integers as weights to emphasize the order of examination of edges. The four cheapest edges are selected, but then we cannot take the edges of weight 5 or 6. We can take the edge of weight 7, but then not those of weight 8 or 9. ■



2.3.3. Theorem. (Kruskal [1956]). In a connected weighted graph G, Kruskal's Algorithm constructs a minimum-weight spanning tree.

Proof: We show first that the algorithm produces a tree. It never chooses an edge that completes a cycle. If the final graph has more than one component, then we considered no edge joining two of them, because such an edge would be accepted. Since G is connected, some such edge exists and we considered it. Thus the final graph is connected and acyclic, which makes it a tree.

Let T be the resulting tree, and let T^* be a spanning tree of minimum weight. It $T = T^*$, we are done. If $T \neq T^*$, let e be the first edge chosen for T that is not in T^*. Adding e to T^* creates one cycle C. Since T has no cycle, C has an edge $e' \notin E(T)$. Consider the spanning tree $T^* + e - e'$.

Since T^* contains e' and all the edges of T chosen before e, both e' and e are available when the algorithm chooses e, and hence $w(e) \leq w(e')$. Thus $T^* + e - e'$ is a spanning tree with weight at most T^* that agrees with T for a longer initial list of edges than T^* does.

Repeating this argument eventually yields a minimum-weight spanning tree that agrees completely with T. Phrased extremally, we have proved that the minimum spanning tree agreeing with T the longest is T itself. ■

2.3.4.* Remark. To implement Kruskal's Algorithm, we first sort the m edge weights. We then maintain for each vertex the label of the component containing it, accepting the next cheapest edge if its endpoints have different labels. We merge the two components by changing the label of each vertex in the smaller component to the label of the larger. Since the size of the component at least doubles when a label changes, each label changes at most $\lg n$ times, and the total number of changes is at most $n \lg n$ (we use lg for the base 2 logarithm).

With this labeling method, the running time for large graphs depends on the time to sort m numbers. With this cost included, other algorithms may be faster than Kruskal's Algorithm. In *Prim's Algorithm* (Exercise 10, due also to Jarník), a spanning tree is grown from a single vertex by iteratively adding the cheapest edge that incorporates a new vertex. Prim's and Kruskal's Algorithms have similar running times when edges are pre-sorted by weight.

Both Borůvka [1926] and Jarník [1930] posed and solved the minimum spanning tree problem. Borůvka's algorithm picks the next edge by considering the cheapest edge leaving each component of the current forest. Modern improvements use clever data structures to merge components quickly. Fast versions appear in Tarjan [1984] for when the edges are pre-sorted and in Gabow–Galil–Spencer–Tarjan [1986] for when they are not. Thorough discussion and further references appear in Ahuja–Magnanti–Orlin [1993, Chapter 13]. More recent developments appear in Karger–Klein–Tarjan [1995]. ■

SHORTEST PATHS

How can we find the shortest route from one location to another? How can we find the shortest routes from our home to every place in town? This requires finding shortest paths from one vertex to all other vertices in a weighted graph. Together, these paths form a spanning tree.

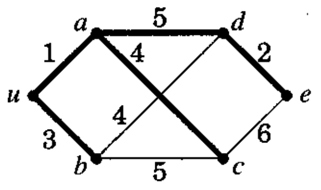
Dijkstra's Algorithm (Dijkstra [1959] and Whiting–Hillier [1960]) solves this problem quickly, using the observation that the u, v-portion of a shortest u, z-path must be a shortest u, v-path. It finds optimal routes from u to other vertices z in increasing order of $d(u, z)$. The **distance** $d(u, z)$ in a weighted graph is the minimum sum of the weights on the edges in a u, z-path (we consider only nonnegative weights).

2.3.5. Algorithm. (Dijkstra's Algorithm—distances from one vertex.)
Input: A graph (or digraph) with nonnegative edge weights and a starting vertex u. The weight of edge xy is $w(xy)$; let $w(xy) = \infty$ if xy is not an edge.
Idea: Maintain the set S of vertices to which a shortest path from u is known, enlarging S to include all vertices. To do this, maintain a tentative distance $t(z)$ from u to each $z \notin S$, being the length of the shortest u, z-path yet found.
Initialization: Set $S = \{u\}$; $t(u) = 0$; $t(z) = w(uz)$ for $z \neq u$.
Iteration: Select a vertex v outside S such that $t(v) = \min_{z \notin S} t(z)$. Add v to S. Explore edges from v to update tentative distances: for each edge vz with $z \notin S$, update $t(z)$ to $\min\{t(z), t(v) + w(vz)\}$.

The iteration continues until $S = V(G)$ or until $t(z) = \infty$ for every $z \notin S$. At the end, set $d(u, v) = t(v)$ for all v. ■

2.3.6. Example. In the weighted graph below, shortest paths from u are found to the other vertices in the order a, b, c, d, e, with distances 1,3,5,6,8, respectively. To reconstruct the paths, we only need the edge on which each shortest path arrives at its destination, because the earlier portion of a shortest u, z-path that reaches z on the edge vz is a shortest u, v-path.

The algorithm can maintain this information by recording the identity of the "selected vertex" whenever the tentative distance to z is updated. When z is selected, the vertex that was recorded when $t(z)$ was last updated is the predecessor of z on the u, z-path of length $d(u, z)$. In this example, the final edges on the paths to a, b, c, d, e generated by the algorithm are ua, ub, ac, ad, de, respectively, and these are the edges of the spanning tree generated from u. ■



With the phrasing given in Algorithm 2.3.5, Dijkstra's Algorithm works also for digraphs, generating an out-tree rooted at u if every vertex is reachable from u. The proof works for graphs and for digraphs. The technique of proving a stronger statement in order to make an inductive proof work is called "loading the induction hypothesis".

2.3.7. Theorem. Given a (di)graph G and a vertex $u \in V(G)$, Dijkstra's Algorithm computes $d(u, z)$ for every $z \in V(G)$.

Proof: We prove the stronger statement that at each iteration,

1) for $z \in S$, $t(z) = d(u, z)$, and

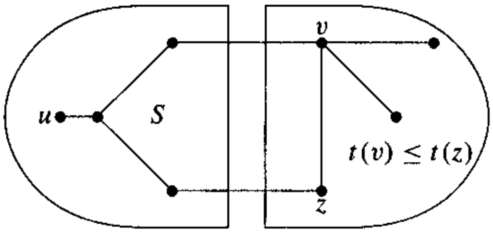
2) for $z \notin S$, $t(z)$ is the least length of a u, z-path reaching z directly from S.

We use induction on $k = |S|$. Basis step: $k = 1$. From the initialization, $S = \{u\}$, $d(u, u) = t(u) = 0$, and the least length of a u, z-path reaching z directly from S is $t(z) = w(u, z)$, which is infinite when uz is not an edge.

Induction step: Suppose that when $|S| = k$, (1) and (2) are true. Let v be a vertex among $z \notin S$ such that $t(z)$ is smallest. The algorithm now chooses v; let $S' = S \cup \{v\}$. We first argue that $d(u, v) = t(v)$. A shortest u, v-path must exit S before reaching v. The induction hypothesis states that the length of the shortest path going directly to v from S is $t(v)$. The induction hypothesis and choice of v also guarantee that a path visiting any vertex outside S and later reaching v has length at least $t(v)$. Hence $d(u, v) = t(v)$, and (1) holds for S'.

To prove (2) for S', let z be a vertex outside S other than v. By the hypothesis, the shortest u, z-path reaching z directly from S has length $t(z)$ (∞ if there is no such path). When we add v to S, we must also consider paths reaching z from v. Since we have now computed $d(u, v) = t(v)$, the shortest such path has length $t(v) + w(vz)$, and we compare this with the previous value of $t(z)$ to find the shortest path reaching z directly from S'.

We have verified that (1) and (2) hold for the new set S' of size $k + 1$; this completes the induction step. ■



The algorithm maintains the condition that $d(u, x) \le t(z)$ for all $x \in S$ and $z \notin S$; hence it selects vertices in nondecreasing order of distance from u. It computes $d(u, v) = \infty$ when v is unreachable from u. The special case for unweighted graphs is **Breadth-First Search** from u. Here both the algorithm and the proof (Exercise 17) have simpler descriptions.

2.3.8. Algorithm. (Breadth-First Search—BFS)
Input: An unweighted graph (or digraph) and a start vertex u.
Idea: Maintain a set R of vertices that have been reached but not searched and a set S of vertices that have been searched. The set R is maintained as a First-In First-Out list (queue), so the first vertices found are the first vertices explored.
Initialization: $R = \{u\}$, $S = \emptyset$, $d(u, u) = 0$.
Iteration: As long as $R \ne \emptyset$, we search from the first vertex v of R. The neighbors of v not in $S \cup R$ are added to the back of R and assigned distance $d(u, v) + 1$, and then v is removed from the front of R and placed in S. ■

The largest distance from a vertex u to another vertex is the eccentricity $\epsilon(u)$. Hence we can compute the diameter of a graph by running Breadth-First Search from each vertex.

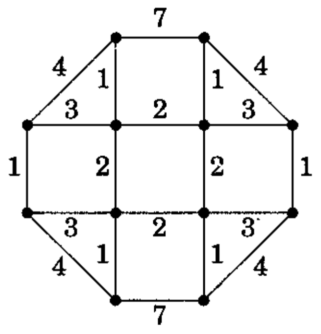
Like Dijkstra's Algorithm, BFS from u yields a tree T in which for each vertex v, the u, v-path is a shortest u, v-path. Thus the graph has no additional edges joining vertices of a u, v-path in T.

Dijkstra's Algorithm figures prominently in the solution of another well-known optimization problem.

2.3.9. Application. A mail carrier must traverse all edges in a road network, starting and ending at the Post Office. The edges have nonnegative weights representing distance or time. We seek a closed walk of minimum total length that uses all the edges. This is the **Chinese Postman Problem**, named in honor of the Chinese mathematician Guan Meigu [1962], who proposed it.

If every vertex is even, then the graph is Eulerian and the answer is the sum of the edge weights. Otherwise, we must repeat edges. Every traversal is an Eulerian circuit of a graph obtained by duplicating edges. Finding the shortest traversal is equivalent to finding the minimum total weight of edges whose duplication will make all vertex degrees even. We say "duplication" because we need not use an edge more than twice. If we use an edge three or more times in making all vertices even, then deleting two of those copies will leave all vertices even. There may be many ways to choose the duplicated edges. ■

2.3.10. Example. In the example below, the eight outer vertices have odd degree. If we match them around the outside to make the degrees even, the extra cost is $4+4+4+4=16$ or $1+7+7+1=16$. We can do better by using all the vertical edges, which total only 10. ■



Adding an edge from an odd vertex to an even vertex makes the even vertex odd. We must continue adding edges until we complete a trail to an odd vertex. The duplicated edges must consist of a collection of trails that pair the odd vertices. We may restrict our attention to paths pairing up the odd vertices (Exercise 24), but the paths may need to intersect.

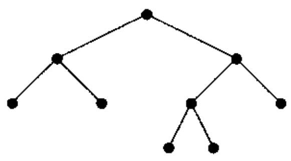
Edmonds and Johnson [1973] described a way to solve the Chinese Postman Problem. If there are only two odd vertices, then we can use Dijkstra's Algorithm to find the shortest path between them and solve the problem. If there are $2k$ odd vertices, then we still can use Dijkstra's Algorithm to find the shortest paths connecting each pair of odd vertices; these are candidates to use in the solution. We use these lengths as weights on the edges of K_{2k}, and then our problem is to find the minimum total weight of k edges that pair up these $2k$ vertices. This is a weighted version of the maximum matching problem discussed in Section 3.3. An exposition appears in Gibbons [1985, p163–165].

TREES IN COMPUTER SCIENCE (optional)

Most applications of trees in computer science use rooted trees.

2.3.11. Definition. A **rooted tree** is a tree with one vertex r chosen as **root**. For each vertex v, let $P(v)$ be the unique v,r-path. The **parent** of v is its neighbor on $P(v)$; its **children** are its other neighbors. Its **ancestors** are the vertices of $P(v)-v$. Its **descendants** are the vertices u such that $P(u)$

contains v. The **leaves** are the vertices with no children. A **rooted plane tree** or **planted tree** is a rooted tree with a left-to-right ordering specified for the children of each vertex.



After a BFS from u, we view the resulting tree T as rooted at u.

2.3.12. Definition. A **binary tree** is a rooted plane tree where each vertex has at most two children, and each child of a vertex is designated as its **left child** or **right child**. The subtrees rooted at the children of the root are the **left subtree** and the **right subtree** of the tree. A k-**ary tree** allows each vertex up to k children.

In many applications of binary trees, all non-leaves have exactly two children (Exercise 26). Binary trees permit storage of data for quick access. We store each item at a leaf and access it by following the path from the root. We encode the path by recording 0 when we move to a left child and 1 when we move to a right child. The search time is the length of this code word for the leaf. Given access probabilities among n items, we want to place them at the leaves of a rooted binary tree to minimize the expected search time.

Similarly, given large computer files and limited storage, we want to encode characters as binary lists to minimize total length. Dividing the frequencies by the total length of the file yields probabilities. This encoding problem then reduces to the problem above.

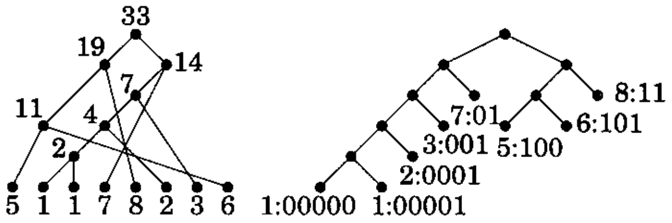
The length of code words may vary; we need a way to recognize the end of the current word. If no code word is an initial portion of another, then the current word ends as soon as the bits since the end of the previous word form a code word. Under this **prefix-free** condition, the binary code words correspond to the leaves of a binary tree using the left/right encoding described above. The expected length of a message is $\sum p_i l_i$, where the ith item has probability p_i and its code has length l_i. Constructing the optimal code is surprisingly easy.

2.3.13. Algorithm. (Huffman's Algorithm [1952]—prefix-free coding)
Input: Weights (frequencies or probabilities) $p_1, \dots, p_n$.
Output: Prefix-free code (equivalently, a binary tree).
Idea: Infrequent items should have longer codes; put infrequent items deeper by combining them into parent nodes.
Initial case: When $n = 2$, the optimal length is one, with 0 and 1 being the codes assigned to the two items (the tree has a root and two leaves; $n = 1$ can also be used as the initial case).

Recursion: When $n > 2$, replace the two least likely items p, p' with a single item q of weight $p + p'$. Treat the smaller set as a problem with $n - 1$ items. After solving it, give children with weights p, p' to the resulting leaf with weight q. Equivalently, replace the code computed for the combined item with its extensions by 1 and 0, assigned to the items that were replaced. ■

2.3.14. Example. *Huffman coding.* Consider eight items with frequencies $5, 1, 1, 7, 8, 2, 3, 6$. Algorithm 2.3.13 combines items according to the tree on the left below, working from the bottom up. First the two items of weight 1 combine to form one of weight 2. Now this and the original item of weight 2 are the least likely and combine to form an item of weight 4. The 3 and 4 now combine, after which the least likely elements are the original items of weights 5 and 6. The remaining combinations in order are $5 + 6 = 11$, $7 + 7 = 14$, $8 + 11 = 19$, and $14 + 19 = 33$.

From the drawing of this tree on the right, we obtain code words. In their original order, the items have code words 100, 00000, 00001, 01, 11, 0001, 001, and 101. The expected length is $\sum p_i l_i = 90/33$. This is less than 3, which would be the expected length of a code using the eight words of length 3. ■



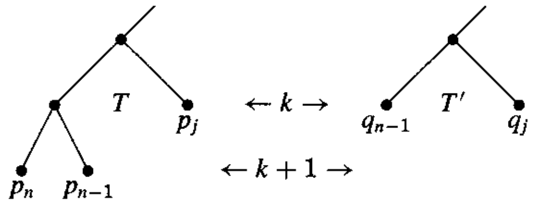
2.3.15. Theorem. Given a probability distribution $\{p_i\}$ on n items, Huffman's Algorithm produces the prefix-free code with minimum expected length.

Proof: We use induction on n. Basis step: $n = 2$. We must send a bit to send a message, and the algorithm encodes each item as a single bit, so the optimum is expected length 1.

Induction step: $n > 2$. Suppose that the algorithm computes the optimal code when given a distribution for $n - 1$ items. Every code assigns items to the leaves of a binary tree. Given a fixed tree with n leaves, we minimize the expected length by greedily assigning the messages with probabilities $p_1 \geq \cdots \geq p_n$ to leaves in increasing order of depth. Thus every optimal code has least likely messages assigned to leaves of greatest depth. Since every leaf at maximum depth has another leaf as its sibling and permuting the items at a given depth does not change the expected length, we may assume that two least likely messages appear as siblings at greatest depth.

Let T be an optimal tree for $p_1, \ldots, p_n$, with the least likely items p_n and p_{n-1} located as sibling leaves at greatest depth. Let T' be the tree obtained from T by deleting these leaves, and let $q_1, \ldots, q_{n-1}$ be the probability distribution obtained by replacing $\{p_{n-1}, p_n\}$ by $q_{n-1} = p_{n-1} + p_n$. The tree T' yields a code for $\{q_i\}$. The expected length for T is the expected length for T' plus q_{n-1}, since if k is the depth of the leaf assigned q_{n-1}, we lose kq_{n-1} and gain $(k+1)(p_{n-1}+p_n)$ in moving from T' to T.

This holds for each choice of T', so it is best to use the tree T' that is optimal for $\{q_i\}$. By the induction hypothesis, the optimal choice for T' is obtained by applying Huffman's Algorithm to $\{q_i\}$. Since the replacement of $\{p_{n-1}, p_n\}$ by q_{n-1} is the first step of Huffman's Algorithm for $\{p_i\}$, we conclude that Huffman's Algorithm generates the optimal tree T for $\{p_i\}$. ■



Huffman's Algorithm computes an optimal prefix-free code, and its expected length is close to the optimum over all types of binary codes. Shannon [1948] proved that for every code with binary digits, the expected length is at least the **entropy** of the discrete probability distribution $\{p_i\}$, defined to be $-\sum p_i \lg p_i$ (Exercise 31). When each p_i is a power of 1/2, the Huffman code meets this bound exactly (Exercise 30).

EXERCISES

2.3.1. (−) Assign integer weights to the edges of K_n. Prove that the total weight on every cycle is even if and only if the total weight on every triangle is even.

2.3.2. (−) Prove or disprove: If T is a minimum-weight spanning tree of a weighted graph G, then the u,v-path in T is a minimum-weight u,v-path in G.

2.3.3. (−) There are five cities in a network. The cost of building a road directly between i and j is the entry $a_{i,j}$ in the matrix below. An infinite entry indicates that there is a mountain in the way and the road cannot be built. Determine the least cost of making all the cities reachable from each other.

$$\begin{pmatrix} 0 & 3 & 5 & 11 & 9 \\ 3 & 0 & 3 & 9 & 8 \\ 5 & 3 & 0 & \infty & 10 \\ 11 & 9 & \infty & 0 & 7 \\ 9 & 8 & 10 & 7 & 0 \end{pmatrix}$$

2.3.4. (−) In the graph below, assign weights $(1,1,2,2,3,3,4,4)$ to the edges in two ways: one way so that the minimum-weight spanning tree is unique, and another way so that the minimum-weight spanning tree is not unique.

2.3.5. (−) There are five cities in a network. The travel time for traveling directly from i to j is the entry $a_{i,j}$ in the matrix below. The matrix is not symmetric (use directed graphs), and $a_{i,j} = \infty$ indicates that there is no direct route. Determine the least travel time and quickest route from i to j for each pair i, j.

$$\begin{pmatrix} 0 & 10 & 20 & \infty & 17 \\ 7 & 0 & 5 & 22 & 33 \\ 14 & 13 & 0 & 15 & 27 \\ 30 & \infty & 17 & 0 & 10 \\ \infty & 15 & 12 & 8 & 0 \end{pmatrix}$$

• • • • •

2.3.6. (!) Assign integer weights to the edges of K_n. Prove that on every cycle the total weight is even if and only if the subgraph consisting of the edges with odd weight is a spanning complete bipartite subgraph. (Hint: Show that every component of the subgraph consisting of the edges with even weight is a complete graph.)

2.3.7. Let G be a weighted connected graph with distinct edge weights. Without using Kruskal's Algorithm, prove that G has only one minimum-weight spanning tree. (Hint: Use Exercise 2.1.34.)

2.3.8. Let G be a weighted connected graph. Prove that no matter how ties are broken in choosing the next edge for Kruskal's Algorithm, the list of weights of a minimum spanning tree (in nondecreasing order) is unique.

2.3.9. Let F be a spanning forest of a connected weighted graph G. Among all edges of G having endpoints in different components of F, let e be one of minimum weight. Prove that among all the spanning trees of G that contain F, there is one of minimum weight that contains e. Use this to give another proof that Kruskal's Algorithm works.

2.3.10. (!) **Prim's Algorithm** grows a spanning tree from a given vertex of a connected weighted graph G, iteratively adding the cheapest edge from a vertex already reached to a vertex not yet reached, finishing when all the vertices of G have been reached. (Ties are broken arbitrarily.) Prove that Prim's Algorithm produces a minimum-weight spanning tree of G. (Jarník [1930], Prim [1957], Dijkstra [1959], independently).

2.3.11. For a spanning tree T in a weighted graph, let $m(T)$ denote the maximum among the weights of the edges in T. Let x denote the minimum of $m(T)$ over all spanning trees of a weighted graph G. Prove that if T is a spanning tree in G with minimum total weight, then $m(T) = x$ (in other words, T also minimizes the maximum weight). Construct an example to show that the converse is false. (Comment: A tree that minimizes the maximum weight is called a **bottleneck** or **minimax** spanning tree.)

2.3.12. In a weighted complete graph, iteratively select the edge of least weight such that the edges selected so far form a disjoint union of paths. After $n - 1$ steps, the result is a spanning path. Prove that this algorithm always gives a minimum-weight spanning path, or give an infinite family of counterexamples where it fails.

2.3.13. (!) Let T be a minimum-weight spanning tree in G, and let T' be another spanning tree in G. Prove that T' can be transformed into T by a list of steps that exchange one edge of T' for one edge of T, such that the edge set is always a spanning tree and the total weight never increases.

2.3.14. (!) Let C be a cycle in a connected weighted graph. Let e be an edge of maximum weight on C. Prove that there is a minimum spanning tree not containing e. Use this to prove that iteratively deleting a heaviest non-cut-edge until the remaining graph is acyclic produces a minimum-weight spanning tree.

2.3.15. Let T be a minimum-weight spanning tree in a weighted connected graph G. Prove that T omits some heaviest edge from every cycle in G.

2.3.16. Four people must cross a canyon at night on a fragile bridge. At most two people can be on the bridge at once. Crossing requires carrying a flashlight, and there is only one flashlight (which can cross only by being carried). Alone, the four people cross in 10, 5, 2, 1 minutes, respectively. When two cross together, they move at the speed of the slower person. In 18 minutes, a flash flood coming down the canyon will wash away the bridge. Can the four people get across in time? Prove your answer without using graph theory and describe how the answer can be found using graph theory.

2.3.17. Given a starting vertex u in an unweighted graph or digraph G, prove directly (without Dijkstra's Algorithm) that Algorithm 2.3.8 computes $d(u, z)$ for all $z \in V(G)$.

2.3.18. Explain how to use Breadth-First Search to compute the girth of a graph.

2.3.19. (+) Prove that the following algorithm correctly finds the diameter of a tree. First, run BFS from an arbitrary vertex w to find a vertex u at maximum distance from w. Next, run BFS from u to reach a vertex v at maximum distance from u. Report diam $T = d(u, v)$. (Cormen–Leiserson–Rivest [1990, p476])

2.3.20. *Minimum diameter spanning tree.* An MDST is a spanning tree where the maximum length of a path is as small as possible. Intuition suggests that running Dijkstra's Algorithm from a vertex of minimum eccentricity (a center) will produce an MDST, but this may fail.

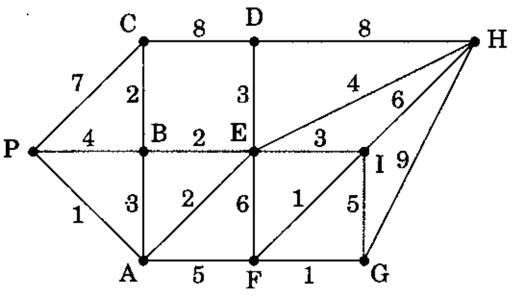
a) Construct a 5-vertex example of an unweighted graph (edge weights all equal 1) such that Dijkstra's Algorithm can be run from some vertex of minimum eccentricity and produce a spanning tree that does not have minimum diameter.

b) Construct a 4-vertex example of a weighted graph such that Dijkstra's algorithm cannot produce an MDST when run from any vertex.

2.3.21. Develop a fast algorithm to test whether a graph is bipartite. The graph is given by its adjacency matrix or by lists of vertices and their neighbors. The algorithm should not need to consider an edge more than twice.

2.3.22. (−) Solve the Chinese Postman Problem in the k-dimensional cube Q_k under the condition that every edge has weight 1.

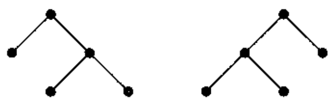
2.3.23. Every morning the Lazy Postman takes the bus to the Post Office. From there, he chooses a route to reach home as quickly as possible (**NOT** ending at the Post Office). Below is a map of the streets along which he must deliver mail, giving the number of minutes required to walk each block whether delivering or not. P denotes the post office and H denotes home. What must the edges traveled more than once satisfy? How many times will each edge be traversed in the optimal route?



2.3.24. (−) Explain why the optimal trails pairing up odd vertices in an optimal solution to the Chinese Postman Problem may be assumed to be paths. Construct a weighted graph with four odd vertices where the optimal solution to the Chinese Postman Problem requires duplicating the edges on two paths that have a common vertex.

2.3.25. Let G be a rooted tree where every vertex has 0 or k children. Given k, for what values of $n(G)$ is this possible?

2.3.26. Find a recurrence relation to count the binary trees with $n+1$ leaves (here each non-leaf vertex has exactly two children, and the left-to-right order of children matters). When $n = 2$, the possibilities are the two trees below.



2.3.27. Find a recurrence relation for the number of rooted plane trees with n vertices. (As in a rooted binary tree, the subtrees obtained by deleting the root of a rooted plane tree are distinguished by their order from left to right.)

2.3.28. (−) Compute a code with minimum expected length for a set of ten messages whose relative frequencies are $1, 2, 3, 4, 5, 5, 6, 7, 8, 9$. What is the expected length of a message in this optimal code?

2.3.29. (−) The game of *Scrabble* has 100 tiles as listed below. This does not agree with English; "S" is less frequent here, for example, to improve the game. Pretend that these are the relative frequencies in English, and compute a prefix-free code of minimum expected length for transmitting messages. Give the answer by listing the relative frequency for each length of code word. Compute the expected length of the code (per text character). (Comment: ASCII coding uses five bits per letter; this code will beat that. Of course, ASCII suffers the handicap of including codes for punctuation.)

A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*L*	*M*	*N*	*O*	*P*	*Q*	*R*	*S*	*T*	*U*	*V*	*W*	*X*	*Y*	*Z*	*Ø*
9	2	2	4	12	2	3	2	9	1	1	4	2	6	8	2	1	6	4	6	4	2	2	1	2	1	2

2.3.30. Consider n messages occurring with probabilities $p_1, \dots, p_n$, such that each p_i is a power of $1/2$ (each $p_i \ge 0$ and $\sum p_i = 1$).

a) Prove that the two least likely messages have equal probability.

b) Prove that the expected message length of the Huffman code for this distribution is $-\sum p_i \lg p_i$.

2.3.31. (+) Suppose that n messages occur with probabilities $p_1, \dots, p_n$ and that the words are assigned distinct binary code words. Prove that for every code, the expected length of a code word with respect to this distribution is at least $-\sum p_i \lg p_i$. (Hint: Use induction on n.) (Shannon [1948])

Chapter 3

Matchings and Factors

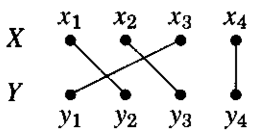
3.1. Matchings and Covers

Within a set of people, some pairs are compatible as roommates; under what conditions can we pair them all up? Many applications of graphs involve such pairings. In Example 1.1.9 we considered the problem of filling jobs with qualified applicants. Bipartite graphs have a natural vertex partition into two sets, and we want to know whether the two sets can be paired using edges. In the roommate question, the graph need not be bipartite.

3.1.1. Definition. A **matching** in a graph G is a set of non-loop edges with no shared endpoints. The vertices incident to the edges of a matching M are **saturated** by M; the others are **unsaturated** (we say *M-saturated* and *M-unsaturated*). A **perfect matching** in a graph is a matching that saturates every vertex.

3.1.2. Example. *Perfect matchings in $K_{n,n}$.* Consider $K_{n,n}$ with partite sets $X = \{x_1, \dots, x_n\}$ and $Y = \{y_1, \dots, y_n\}$. A perfect matching defines a bijection from X to Y. Successively finding mates for $x_1, x_2, \dots$ yields $n!$ perfect matchings.

Each matching is represented by a permutation of $[n]$, mapping i to j when x_i is matched to y_j. We can express the matchings as matrices. With X and Y indexing the rows and columns, we let position i, j be 1 for each edge $x_i y_j$ in a matching M to obtain the corresponding matrix. There is one 1 in each row and each column. ■

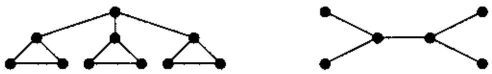


$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3.1.3. Example. *Perfect matchings in complete graphs.* Since it has odd order, K_{2n+1} has no perfect matching. The number f_n of perfect matchings in K_{2n} is the number of ways to pair up $2n$ distinct people. There are $2n - 1$ choices for the partner of v_{2n}, and for each such choice there are f_{n-1} ways to complete the matching. Hence $f_n = (2n - 1)f_{n-1}$ for $n \geq 1$. With $f_0 = 1$, it follows by induction that $f_n = (2n - 1) \cdot (2n - 3) \cdots (1)$.

There is also a counting argument for f_n. From an ordering of $2n$ people, we form a matching by pairing the first two, the next two, and so on. Each ordering thus yields one matching. Each matching is generated by $2^n n!$ orderings, since change the order of the pairs or the order within a pair does not change the resulting matching. Thus there are $f_n = (2n)!/(2^n n!)$ perfect matchings. ■

The usual drawing of the Petersen graph shows a perfect matching and two 5-cycles; counting the perfect matchings takes some effort (Exercise 14). The inductive construction of the hypercube Q_k readily yields many perfect matchings (Exercise 16), but counting them exactly is difficult. The graphs below have even order but no perfect matchings.



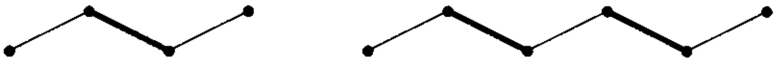
MAXIMUM MATCHINGS

A matching is a set of edges, so its **size** is the number of edges. We can seek a large matching by iteratively selecting edges whose endpoints are not used by the edges already selected, until no more are available. This yields a maximal matching but maybe not a maximum matching.

3.1.4. Definition. A **maximal matching** in a graph is a matching that cannot be enlarged by adding an edge. A **maximum matching** is a matching of maximum size among all matchings in the graph.

A matching M is maximal if every edge not in M is incident to an edge already in M. Every maximum matching is a maximal matching, but the converse need not hold.

3.1.5. Example. *Maximal $\neq$ maximum.* The smallest graph having a maximal matching that is not a maximum matching is P_4. If we take the middle edge, then we can add no other, but the two end edges form a larger matching. Below we show this phenomenon in P_4 and in P_6. ■



In Example 3.1.5, replacing the bold edges by the solid edges yields a larger matching. This gives us a way to look for larger matchings.

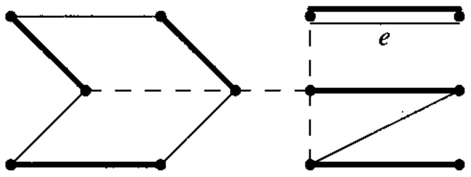
3.1.6. Definition. Given a matching M, an M**-alternating path** is a path that alternates between edges in M and edges not in M. An M-alternating path whose enpoints are unsaturated by M is an M**-augmenting path**.

Given an M-augmenting path P, we can replace the edges of M in P with the other edges of P to obtain a new matching M' with one more edge. Thus when M is a maximum matching, there is no M-augmenting path.

In fact, we prove next that maximum matchings are characterized by the absence of augmenting paths. We prove this by considering two matchings and examining the set of edges belonging to exactly one of them. We define this operation for any two graphs with the same vertex set. (The operation is defined in general for any two sets; see Appendix A.)

3.1.7. Definition. If G and H are graphs with vertex set V, then the **symmetric difference** $G \triangle H$ is the graph with vertex set V whose edges are all those edges appearing in exactly one of G and H. We also use this notation for sets of edges; in particular, if M and M' are matchings, then $M \triangle M' = (M - M') \cup (M' - M)$.

3.1.8. Example. In the graph below, M is the matching with five solid edges, M' is the one with six bold edges, and the dashed edges belong to neither M nor M'. The two matchings have one common edge e; it is not in their symmetric difference. The edges of $M \triangle M'$ form a cycle of length 6 and a path of length 3. ■



3.1.9. Lemma. Every component of the symmetric difference of two matchings is a path or an even cycle.

Proof: Let M and M' be matchings, and let $F = M \triangle M'$. Since M and M' are matchings, every vertex has at most one incident edge from each of them. Thus F has at most two edges at each vertex. Since $\Delta(F) \leq 2$, every component of F is a path or a cycle. Furthermore, every path or cycle in F alternates between edges of $M - M'$ and edges of $M' - M$. Thus each cycle has even length, with an equal number of edges from M and from M'. ■

3.1.10. Theorem. (Berge [1957]) A matching M in a graph G is a maximum matching in G if and only if G has no M-augmenting path.

Proof: We prove the contrapositive of each direction; G has a matching larger than M if and only if G has an M-augmenting path. We have observed that an M-augmenting path can be used to produce a matching larger than M.

For the converse, let M' be a matching in G larger than M; we construct an M-augmenting path. Let $F = M \triangle M'$. By Lemma 3.1.9, F consists of paths and even cycles; the cycles have the same number of edges from M and M'. Since $|M'| > |M|$, F must have a component with more edges of M' than of M. Such a component can only be a path that starts and ends with an edge of M'; thus it is an M-augmenting path in G. ■

HALL'S MATCHING CONDITION

When we are filling jobs with applicants, there may be many more applicants than jobs; successfully filling the jobs will not use all applicants. To model this problem, we consider an X,Y-bigraph (bipartite graph with bipartition X,Y—Definition 1.2.17), and we seek a matching that saturates X.

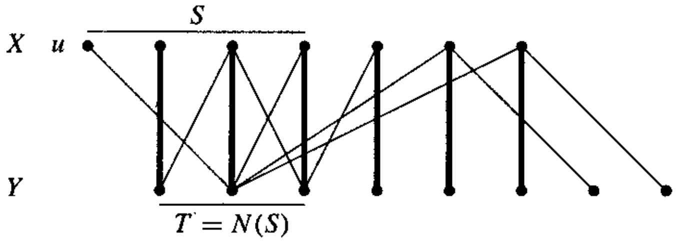
If a matching M saturates X, then for every $S \subseteq X$ there must be at least $|S|$ vertices that have neighbors in S, because the vertices matched to S must be chosen from that set. We use $N_G(S)$ or simply $N(S)$ to denote the set of vertices having a neighbor in S. Thus $|N(S)| \geq |S|$ is a necessary condition.

The condition "For all $S \subseteq X$, $|N(S)| \geq |S|$" is **Hall's Condition**. Hall proved that this obvious necessary condition is also sufficient (TONCAS).

3.1.11. Theorem. (Hall's Theorem—P. Hall [1935]) An X,Y-bigraph G has a matching that saturates X if and only if $|N(S)| \geq |S|$ for all $S \subseteq X$.

Proof: *Necessity*. The $|S|$ vertices matched to S must lie in $N(S)$.

Sufficiency. To prove that Hall's Condition is sufficient, we prove the contrapositive. If M is a maximum matching in G and M does not saturate X, then we obtain a set $S \subseteq X$ such that $|N(S)| < |S|$. Let $u \in X$ be a vertex unsaturated by M. Among all the vertices reachable from u by M-alternating paths in G, let S consist of those in X, and let T consist of those in Y (see figure below with M in bold). Note that $u \in S$.



We claim that M matches T with $S - \{u\}$. The M-alternating paths from u reach Y along edges not in M and return to X along edges in M. Hence every vertex of $S - \{u\}$ is reached by an edge in M from a vertex in T. Since there is no M-augmenting path, every vertex of T is saturated; thus an M-alternating