Combinatorial Optimization

Problem set 6: solutions

1. Suppose a simple undirected graph has more than one minimum spanning tree. Can Prim's algorithm (or Kruskal's algorithm) be used to find all of them? Explain why or why not, and give an example.

▷ **Solution.** Yes, Prim's algorithm (or Kruskal's algorithm) can be used to find all minimum spanning trees. Of course, each of these algorithms only produces a single minimum spanning tree when it is run, so what we really mean is that every minimum spanning tree is a possible result of both of these algorithms; by making the appropriate choices during the algorithm, any particular minimum spanning tree can be found.

We begin with a lemma whose proof is essentially the same as the proof of Theorem 12.1 in Papadimitriou and Steiglitz.

Lemma. *Let $G = (V, E)$ be a connected graph with an edge weight $d(u, v)$ for each edge $\{u, v\} \in E$. Let (V, T) be a minimum spanning tree of G, and let (V, F) be a spanning forest such that $F \subsetneq T$. Let $U \subsetneq V$ be the vertex set of a connected component of (V, F), and let $\partial U \subseteq E$ be the set of edges of G having exactly one endpoint in U. (The set ∂U is called the* boundary *of U.) Then among the edges in ∂U having minimum weight, there is one that is in T.*

Proof. Suppose for the sake of contradiction that the set of minimum-weight edges in ∂U contains no edges of T. Let $\{u, v\} \in \partial U$ be a minimum-weight edge. Add $\{u, v\}$ to T; this creates a (unique) cycle. Because this cycle contains at least one vertex in U and at least one vertex not in U (namely, u and v), it must contain another edge $\{u', v'\} \in \partial U$ different from $\{u, v\}$, and $\{u', v'\}$ must be in T. By assumption, $d(u, v) < d(u', v')$, so if we remove the edge $\{u', v'\}$ we obtain a new spanning tree (V, T'), with $T' = T \cup \big\{\{u, v\}\big\} \setminus \big\{\{u', v'\}\big\}$, with strictly smaller weight than (V, T). But this contradicts the fact that (V, T) is a minimum spanning tree. ∎

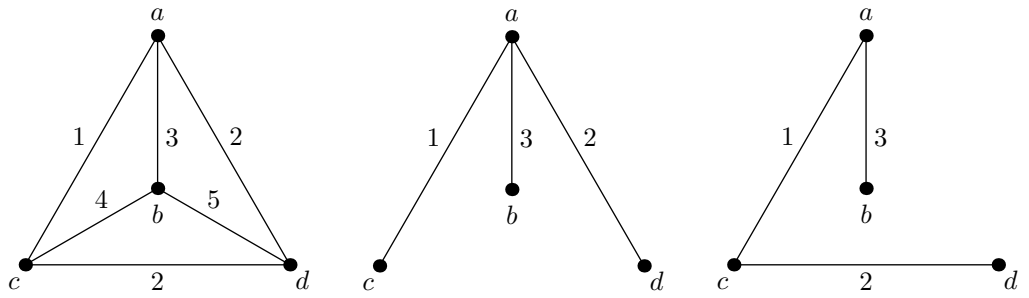
Each iteration of Prim's algorithm (or Kruskal's algorithm) consists of adding an edge to a spanning forest, initially having no edges, until the spanning forest becomes a spanning tree. Therefore, the spanning forest that results after k iterations of either of these algorithms has exactly k edges.

Claim. *Fix a minimum spanning tree (V, T) of G. For every $0 \le k \le n - 1$, after k iterations of Prim's algorithm (or Kruskal's algorithm), it is possible for the resulting spanning forest (V, F) to satisfy $F \subseteq T$.*

Proof. By induction on k. Clearly the statement is true for $k = 0$, because both Prim's algorithm and Kruskal's algorithm begin with a spanning forest having no edges, so $F = \emptyset \subseteq T$. Suppose $k \ge 1$. By induction, it is possible, after $k-1$ iterations, for Prim's algorithm (or Kruskal's algorithm) to produce a spanning forest (V, F') having $F' \subseteq T$; since $k-1 < n-1$, we know that $F' \ne T$. In the kth iteration of Prim's algorithm, a minimum-weight edge e is selected out of the set ∂U of edges having exactly one endpoint in U, where U is the vertex set of the connected component of (V, F') containing a fixed vertex v_1 arbitrarily chosen at the beginning of the algorithm. In the kth iteration of Kruskal's algorithm, a minimum-weight edge $e = \{u, v\}$ is selected out of the set of all edges whose endpoints are in different connected components of (V, F'); this edge is in the set ∂U, where U is the vertex set of the connected component of (V, F') containing u. In either case, we see by the lemma above that among the edges in ∂U having minimum weight, there is one that is in T, so we may select e to be an edge in T, and then $F := F' \cup \{e\} \subseteq T$. ∎

Therefore, taking $k = n-1$ in this claim, we see that it is possible for Prim's algorithm (or Kruskal's algorithm) to produce the tree (V, T). As (V, T) was an arbitrary minimum spanning tree, this shows that it is possible for both of these algorithms to find any minimum spanning tree.

As an example, consider the graph shown on the left below. It has two minimum spanning trees, shown separately.



If v_1 is chosen to be the vertex a at the beginning of Prim's algorithm, then both Prim's algorithm and Kruskal's algorithm start by selecting the edge $\{a, c\}$. In the next iteration, both $\{a, d\}$ and $\{c, d\}$ are candidates to be selected. Selecting $\{a, d\}$ leads to the minimum spanning tree shown in the center above, while selecting $\{c, d\}$ leads to the one shown on the right. □

2. Explain how a minimum spanning tree algorithm can easily be used to find a *maximum* spanning tree in a graph. Then explain why, if you want to find the *longest* path between two vertices in a graph, using this same technique with Dijkstra's algorithm does not work.

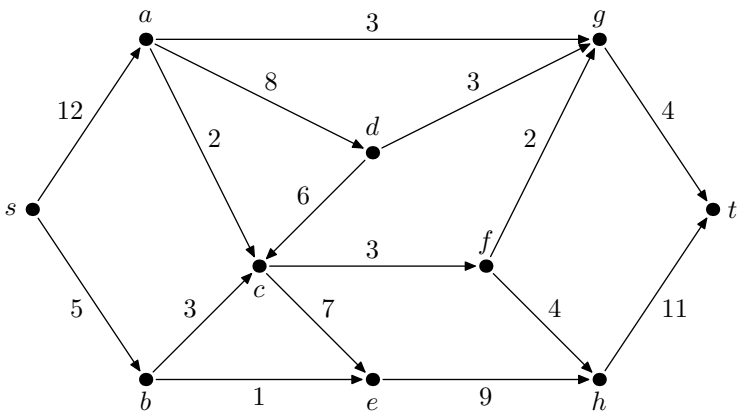
▷ **Solution.** A maximum spanning tree can be found by negating all of the edge weights and then applying a minimum spanning tree algorithm to the resulting graph.

Alternatively, if we would like to work only with nonnegative edge weights, we can find the largest edge weight M, replace every edge weight w with $M - w$, and then apply a minimum spanning tree algorithm. This is equivalent to negating all of the edge weights and then increasing each edge weight by M. This approach works because every spanning tree on a graph with n vertices has exactly $n-1$ edges, so the total weight of every spanning tree will be increased by exactly $(n-1)M$.

The simple reason that negating edge weights does not work to find a longest path with Dijkstra's algorithm is that Dijkstra's algorithm requires all weights to be nonnegative.

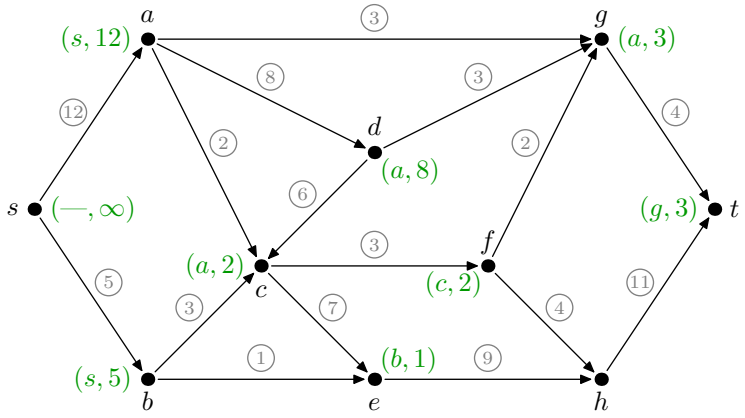
The $M - w$ trick doesn't work with Dijkstra's algorithm either, because different paths between a given pair of vertices may consist of different numbers of edges. Negating each edge weight and then adding M to each weight will therefore add different multiples of M to the total weights of these paths, and this will not necessarily preserve the longest or shortest path. □

3. Find a maximum s–t flow and a minimum s–t cut in the following flow network.



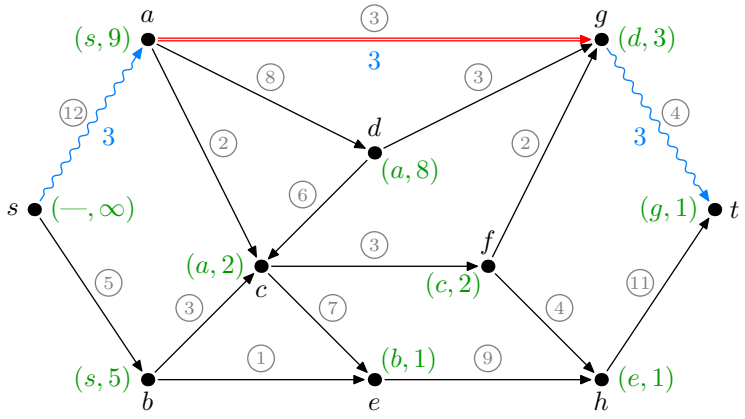
▷ **Solution.** The iterations of the Ford–Fulkerson algorithm are shown below. In these figures, the capacities of the arcs are shown circled in gray, the node labels are in green,

and nonzero flows along arcs are in light blue. The node labels are given in the form $(from[i], how\text{-}much[i])$. Arcs with zero flow are drawn as straight black arrows, saturated arcs are drawn as double red arrows, and unsaturated arcs with nonzero flow are drawn as wavy light blue arrows. Below the figure is the *LIST* of vertices built up as the Ford–Fulkerson algorithm progresses through the network. We treat the *LIST* as a queue here: nodes are added to the right-hand side when they are labeled, and they are removed from the left-hand side (indicated by crossing them out) when they are scanned.



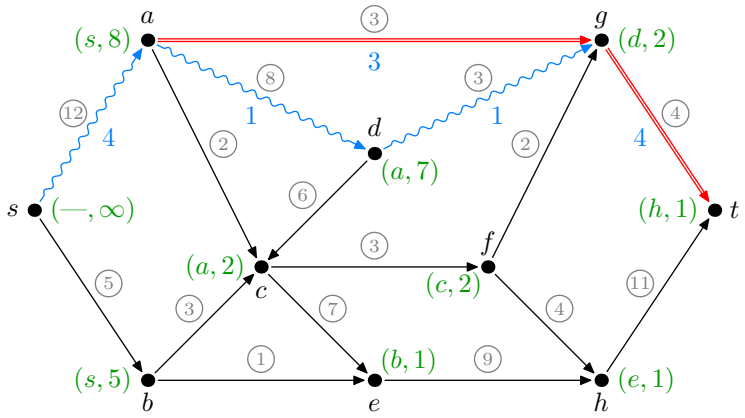
LIST: ~~s~~; ~~a~~, ~~b~~; ~~c~~, ~~d~~, ~~g~~; e; f; t.

The node t received a label, so we have found an augmenting path. The value of the label $how\text{-}much[t]$ is 3, so we can augment the flow by 3 along this path. To identify the path, we follow the *from* labels backward from t: $from[t] = g$, $from[g] = a$, and $from[a] = s$, so our augmenting path is s–a–g–t. All of the *from* labels along this path were "positive" (they did not have a leading minus sign), so the flow along every arc in this path will be increased by 3. We adjust the flow and repeat the labeling process from the beginning.



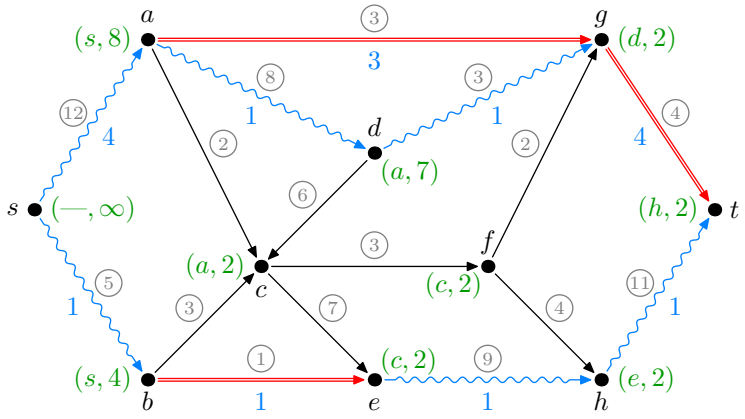
LIST: ~~s~~; ~~a~~, ~~b~~; ~~c~~, ~~d~~; ~~e~~; ~~f~~; ~~g~~; h; t.

Following the *from* labels backward from t, we find the augmenting path s–a–d–g–t. We augment the flow along every arc in this path by $how\text{-}much[t] = 1$ and repeat the labeling process from the beginning.



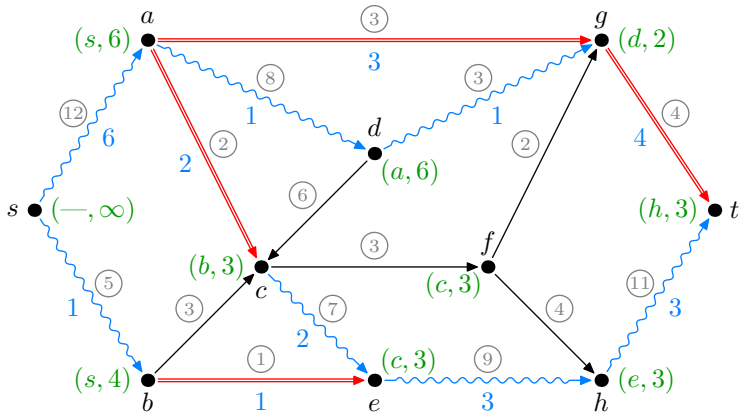
LIST: ~~s~~; ~~a~~, ~~b~~; ~~c~~, ~~d~~; ~~e~~; ~~f~~; ~~g~~; ~~h~~; *t*.

The augmenting path here is *s*–*b*–*e*–*h*–*t*; we augment the flow along this path by 1.



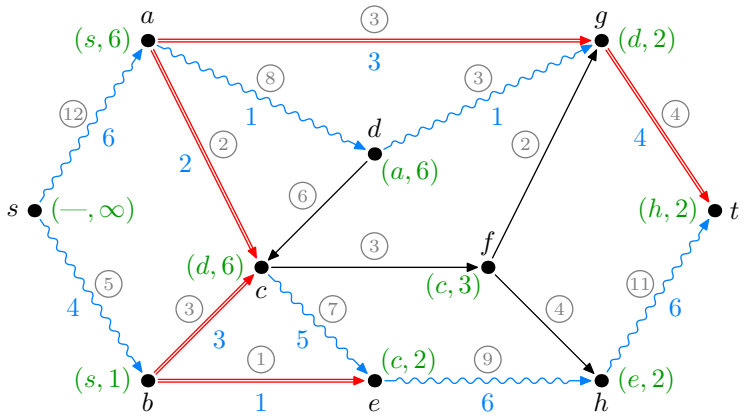
LIST: ~~s~~; ~~a~~, ~~b~~; ~~c~~, ~~d~~; ~~e~~, ~~f~~; ~~g~~; ~~h~~; *t*.

The augmenting path here is *s*–*a*–*c*–*e*–*h*–*t*; we augment the flow along this path by 2.



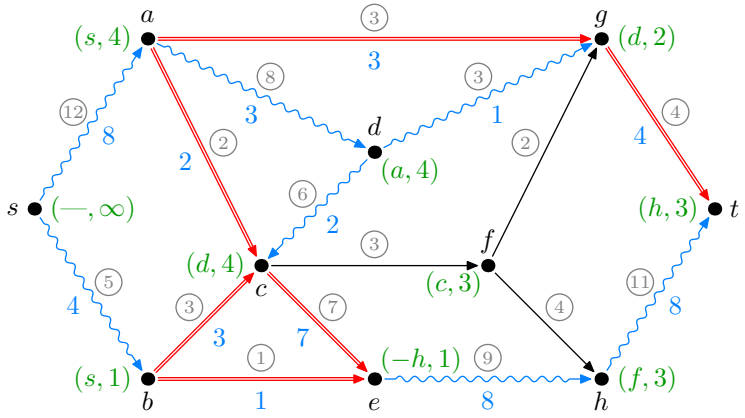
LIST: ~~s~~; ~~a~~, ~~b~~; ~~d~~; ~~c~~; ~~g~~; ~~e~~, ~~f~~; ~~h~~; *t*.

The augmenting path here is *s*–*b*–*c*–*e*–*h*–*t*; we augment the flow along this path by 3.



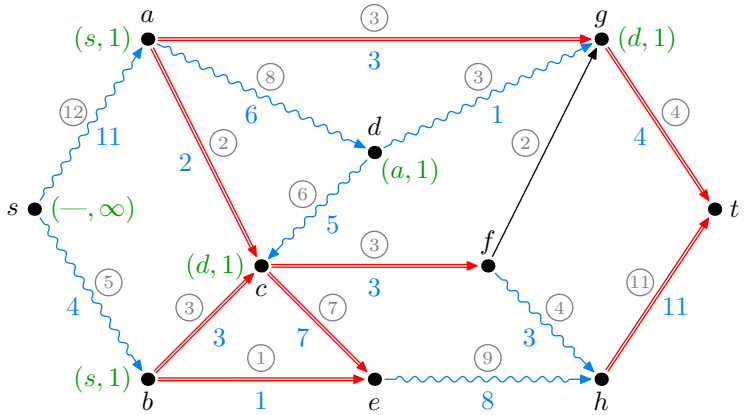
LIST: ~~s~~; ~~a~~, ~~b~~; ~~d~~; ~~c~~, ~~g~~; ~~e~~, ~~f~~; ~~h~~; t.

The augmenting path here is s–a–d–c–e–h–t; we augment the flow along this path by 2.



LIST: ~~s~~; ~~a~~, ~~b~~; ~~d~~; ~~c~~, ~~g~~; ~~f~~; ~~h~~; e, t.

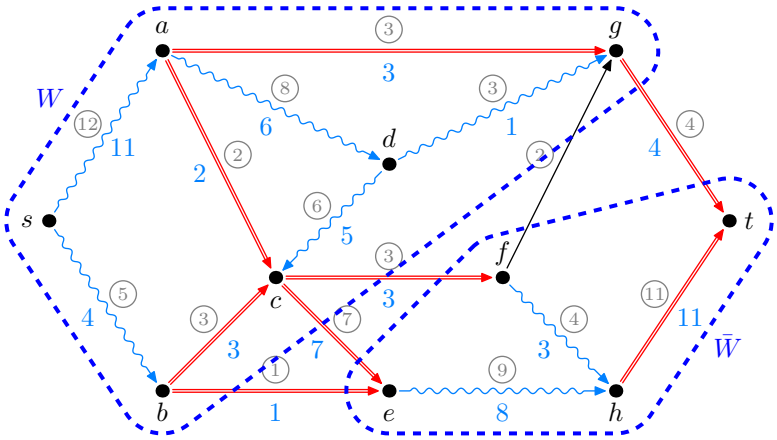
The augmenting path here is s–a–d–c–f–h–t; we augment the flow along this path by 3.



LIST: ~~s~~; ~~a~~, ~~b~~; ~~d~~; ~~c~~, ~~g~~.

Now t did not get a label, so there is no augmenting path, so this flow is optimal. The value of this flow is 15. A minimum s–t cut is determined by partitioning the nodes according to whether they received a label in this last iteration:

$$W = \{s, a, b, c, d, g\}, \qquad \bar{W} = \{e, f, h, t\}.$$



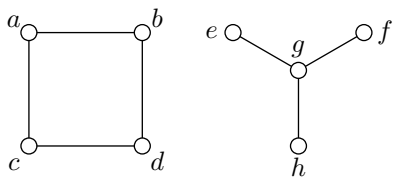
The capacity of this cut is the sum of the capacities b_{ij} of the arcs that point from a node i in W to a node j in $\bar{W}$: $b_{be} + b_{ce} + b_{cf} + b_{gt} = 1 + 7 + 3 + 4 = 15$, which equals the value of the maximum s–t flow, in accordance with the max-flow min-cut theorem. Note also that in the maximum flow all arcs that go forward across the cut (from W to $\bar{W}$) are saturated while all arcs that go backward across the cut (from $\bar{W}$ to W) have zero flow. □

4. Carefully describe an algorithm for the following problem: Given a simple undirected graph $G = (V, E)$, determine whether G is bipartite, and if so give a bipartition (i.e., a partition of the vertex set V into two sets U and W such that every edge has one endpoint in U and one endpoint in W). Illustrate the operation of your algorithm on two examples, one bipartite graph and one non-bipartite graph. Prove that your algorithm is correct in general.

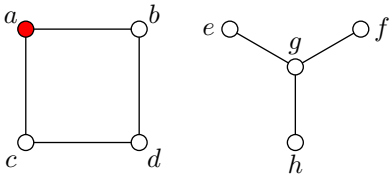
▷ **Solution.** Here are the steps of an algorithm for this problem.

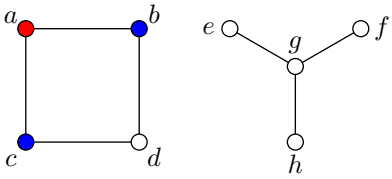
1. Begin with all vertices uncolored. Set $LIST := \emptyset$.
2. Choose an uncolored vertex v. Color v red. Add v to $LIST$.
3. Choose a vertex w in $LIST$, and remove w from $LIST$.
4. If any neighbor of w has the same color as w, stop: the graph is not bipartite.
5. For each uncolored neighbor x of w: color x blue if w is red, or color x red if w is blue; and then add x to $LIST$.
6. If $LIST$ is nonempty, go back to step 3.
7. If there still exist uncolored vertices, go back to step 2.
8. We are done. The graph is bipartite. Let U be the set of red vertices, and let W be the set of blue vertices; then (U, W) is a bipartition.

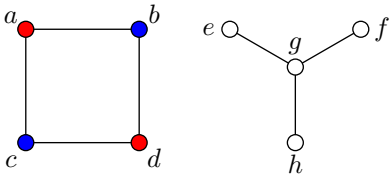
For example, consider the following graph, which is bipartite:

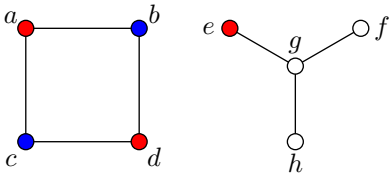


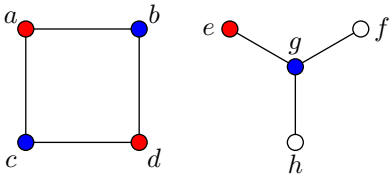
1. We begin with all vertices uncolored and an empty $LIST$.
2. We choose an uncolored vertex, say a, and color it red. We add a to $LIST$. Now $LIST = \{a\}$.
3. We choose a vertex in $LIST$; it must be a, as that is the only element of $LIST$. We remove it from $LIST$, so now $LIST$ is empty again.
4. No neighbor of a has the same color as a, so we skip step 4.
5. We color each uncolored neighbor of a blue and add these vertices to $LIST$. These vertices are b and c. Now $LIST = \{b, c\}$.
6. $LIST$ is nonempty, so we go back to step 3.
3. We choose a vertex in $LIST$, say b, and remove it from $LIST$. Now $LIST = \{c\}$.
4. No neighbor of b has the same color as b, so we skip step 4.
5. We color each uncolored neighbor of b red and add these vertices to $LIST$. This is only the vertex d. Now $LIST = \{c, d\}$.
6. $LIST$ is nonempty, so we go back to step 3.
3. We choose a vertex in $LIST$, say c, and remove it from $LIST$. Now $LIST = \{d\}$.
4. No neighbor of c has the same color as c, so we skip step 4.
5. No neighbor of c is uncolored, so there is nothing to do in step 5.
6. $LIST$ is nonempty, so we go back to step 3.
3. We choose a vertex in $LIST$, which must be d, and remove it from $LIST$. Now $LIST$ is empty.
4. No neighbor of d has the same color as d, so we skip step 4.
5. No neighbor of d is uncolored, so there is nothing to do in step 5.
6. $LIST$ is empty, so we skip step 6.
7. There still exist uncolored vertices, so we go back to step 2.
2. We choose an uncolored vertex, say e, and color it red. We add e to $LIST$. Now $LIST = \{e\}$.
3. We choose a vertex in $LIST$, which must be e, and remove it from $LIST$. Now $LIST$ is empty.
4. No neighbor of e has the same color as e, so we skip step 4.
5. We color each uncolored neighbor of e blue and add these vertices to $LIST$. This is only the vertex g. Now $LIST = \{g\}$.
6. $LIST$ is nonempty, so we go back to step 3.
3. We choose a vertex in $LIST$, which must be g, and remove it from $LIST$. Now $LIST$ is empty.



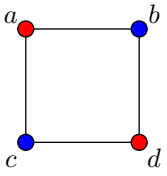


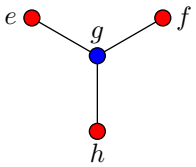




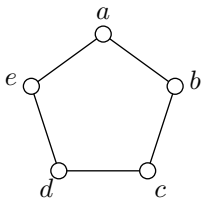


4. No neighbor of g has the same color as g, so we skip step 4.
5. We color each uncolored neighbor of g red and add these vertices to *LIST*. These vertices are f and h. Now $LIST = \{f, h\}$.
6. *LIST* is nonempty, so we go back to step 3.
3. We choose a vertex in *LIST*, say f, and remove it from *LIST*. Now $LIST = \{h\}$.
4. No neighbor of f has the same color as f, so we skip step 4.
5. No neighbor of f is uncolored, so there is nothing to do in step 5.
6. *LIST* is nonempty, so we go back to step 3.
3. We choose a vertex in *LIST*, which must be h, and remove it from *LIST*. Now *LIST* is empty.
4. No neighbor of h has the same color as h, so we skip step 4.
5. No neighbor of h is uncolored, so there is nothing to do in step 5.
6. *LIST* is empty, so we skip step 6.
7. All vertices have been colored, so we skip step 7.
8. The graph is bipartite, and $U = \{a, d, e, f, h\}$, $W = \{b, c, g\}$ is a bipartition. ✓

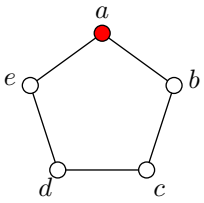


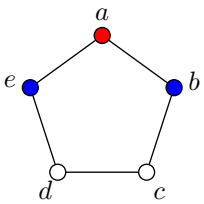


For another example, consider the following graph, which is not bipartite:

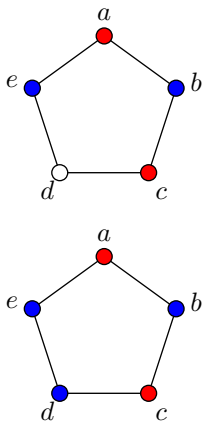


1. We begin with all vertices uncolored and an empty *LIST*.
2. We choose an uncolored vertex, say a, and color it red. We add a to *LIST*. Now $LIST = \{a\}$.
3. We choose a vertex in *LIST*, which must be a, and remove it from *LIST*. Now *LIST* is empty.
4. No neighbor of a has the same color as a, so we skip step 4.
5. We color each uncolored neighbor of a blue and add these vertices to *LIST*. These vertices are b and e. Now $LIST = \{b, e\}$.
6. *LIST* is nonempty, so we go back to step 3.
3. We choose a vertex in *LIST*, say b, and remove it from *LIST*. Now $LIST = \{e\}$.
4. No neighbor of b has the same color as b, so we skip step 4.





5. We color each uncolored neighbor of b red and add these vertices to $LIST$. This is only the vertex c. Now $LIST = \{c, e\}$.
6. $LIST$ is nonempty, so we go back to step 3.
3. We choose a vertex in $LIST$, say c, and remove it from $LIST$. Now $LIST = \{e\}$.
4. No neighbor of c has the same color as c, so we skip step 4.
5. We color each uncolored neighbor of c blue and add these vertices to $LIST$. This is only the vertex d. Now $LIST = \{d, e\}$.
6. $LIST$ is nonempty, so we go back to step 3.
3. We choose a vertex in $LIST$, say d, and remove it from $LIST$. Now $LIST = \{e\}$.
4. We notice that one of the neighbors of d, namely e, has the same color as d. Therefore we report that the graph is not bipartite. ✓



In order to prove the correctness of this algorithm, we first prove Proposition 1 from Section A.2 of Papadimitriou and Steiglitz, and in order to do that we first prove a lemma.

Lemma. *Every closed odd walk contains an odd cycle.*

Proof. By induction on the length l of the walk.

The base case for general graphs is $l = 1$. In this case, the walk consists of a loop (an edge joining a vertex to itself). The loop itself is an odd cycle. Note that this cannot occur in a *simple* graph, which has no loops.

The base case for simple graphs is $l = 3$. In this case, the walk must be a cycle of length 3 (because the graph does not contain loops).

For the inductive step, suppose $l \geq 5$ (or $l \geq 3$ for general graphs). If no vertex is repeated in the walk (except the first and last vertex, which are the same), then the walk itself is an odd cycle. Otherwise, some vertex v is repeated, and we can break the walk into two v–v walks. Since l is odd, the length of one of these v–v walks must be odd, and by induction it contains an odd cycle. ∎

Proposition 1. *A graph is bipartite iff it has no circuit of odd length.*

Proof. ($\Longrightarrow$) Suppose G is bipartite, and let (U, W) be a bipartition of the vertex set. Then every edge has one endpoint in U and one endpoint in W, so the vertices of a walk alternate between vertices in U and vertices in W. Thus a walk that begins and ends at the same vertex (in particular, a cycle) must have even length. So G has no cycles of odd length.

($\Longleftarrow$) Suppose $G = (V, E)$ has no cycles of odd length. We shall construct a bipartition (U, W) of G. Without loss of generality, we may assume that G is connected; otherwise, we can construct a bipartition (U_i, W_i) of each connected component separately and then take $U = \bigcup U_i$ and $W = \bigcup W_i$.

Fix an arbitrary vertex $u \in V$, and for all $v \in V$ define $d(v)$ to be the shortest length of a path from u to v. [Since G is connected by assumption, $d(v)$ is well defined for all $v \in V$.] Let $U = \{\, v \in V : \mathrm{d(v) \text{ is even}} \,\}$ and $W = \{\, v \in V : \mathrm{d(v) \text{ is odd}} \,\}$. If there is an edge between two vertices $u_1, u_2 \in U$, then the shortest path from u to u_1, plus the edge from u_1 to u_2, plus the shortest path from u_2 to u yields a closed odd walk in U. By the preceding lemma, such a walk contains an odd cycle. But this contradicts the hypothesis that G contains no odd cycles, so there can be no such edge between vertices in U. Likewise, there can be no edge between vertices in W. So (U, W) is a bipartition. ∎

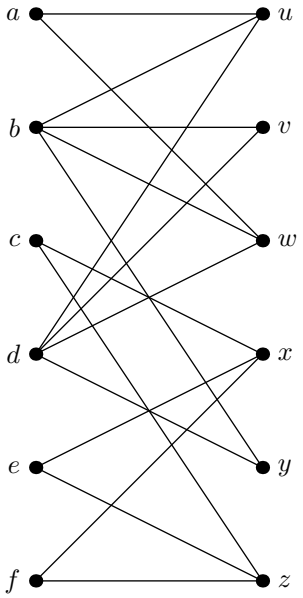
Now we establish the correctness of the algorithm.

Suppose the algorithm reports that the graph is bipartite. This must happen in step 8, which cannot happen until all vertices have been colored and processed by step 4. If the resulting vertex coloring of the graph assigns the same color to both endpoints of an edge, then this would have been noticed in step 4 when processing the second endpoint; so this

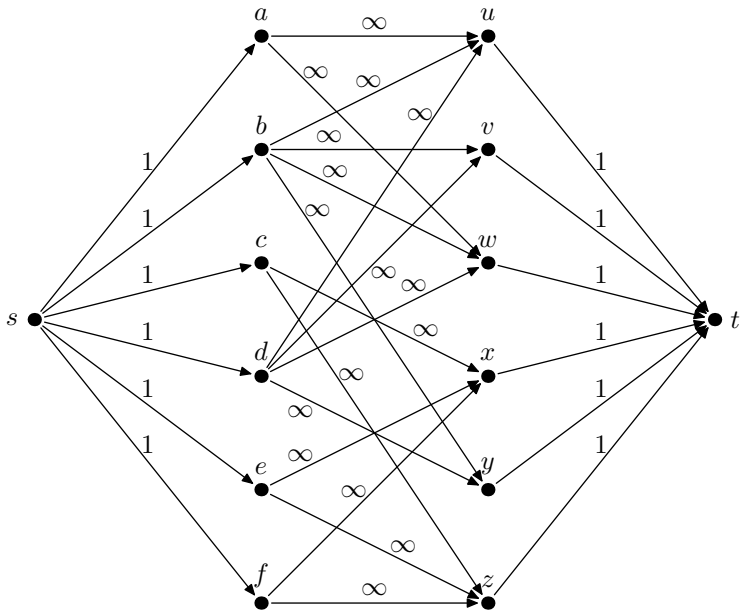
does not happen. Therefore every edge has one red endpoint and one blue endpoint, so (U, W) is a bipartition, and the graph is indeed bipartite.

On the other hand, suppose the algorithm reports that the graph is not bipartite. this occurs when it is noticed in step 4 that two adjacent vertices w and x have the same color. The vertices chosen as v in step 2 are the first vertices in their connected components to be assigned a color. Every other vertex is assigned red if it is an even distance from one of these v's or blue if it is an odd distance (where "distance" means "the number of edges in a shortest path"). If two adjacent vertices have the same color, then there is a closed odd walk in the graph [as in the ($\Longleftarrow$) part of the proof of Proposition 1], so the graph contains an odd cycle, so it is indeed not bipartite. □

5. Find a maximum matching in the following bipartite graph.



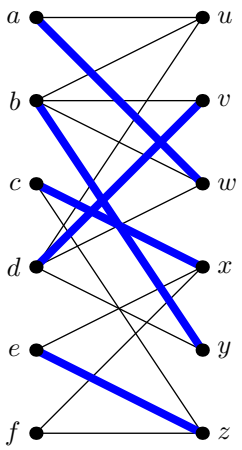
▷ **Solution.** One way to do this is to reduce the problem to that of finding a maximum s–t flow in the following network:



The corresponding max-flow LP is shown below. In this LP, the variable v denotes the value of the flow, and for each arc (i, j) in the network the variable f_{ij} denotes the flow along that arc. The first 14 constraints enforce flow balance at every node: the net outflow at every node must be 0, except at node s where it is v and at node t where it is $-v$. The remaining 12 constraints enforce the capacities of the arcs from s and the arcs to t.

$$
\begin{aligned}
\text{maximize} \quad & v \\
\text{subject to} \quad & f_{sa} + f_{sb} + f_{sc} + f_{sd} + f_{se} + f_{sf} = v \\
& f_{au} + f_{aw} - f_{sa} = 0 \\
& f_{bu} + f_{bv} + f_{bw} + f_{by} - f_{sb} = 0 \\
& f_{cx} + f_{cz} - f_{sc} = 0 \\
& f_{du} + f_{dv} + f_{dw} + f_{dy} - f_{sd} = 0 \\
& f_{ex} + f_{ez} - f_{se} = 0 \\
& f_{fx} + f_{fz} - f_{sf} = 0 \\
& f_{ut} - f_{au} - f_{bu} - f_{du} = 0 \\
& f_{vt} - f_{bv} - f_{dv} = 0 \\
& f_{wt} - f_{aw} - f_{bw} - f_{dw} = 0 \\
& f_{xt} - f_{cx} - f_{ex} - f_{fx} = 0 \\
& f_{yt} - f_{by} - f_{dy} = 0 \\
& f_{zt} - f_{cz} - f_{ez} - f_{fz} = 0 \\
& -f_{ut} - f_{vt} - f_{wt} - f_{xt} - f_{yt} - f_{zt} = -v \\
& f_{sa} \le 1 \qquad f_{ut} \le 1 \\
& f_{sb} \le 1 \qquad f_{vt} \le 1 \\
& f_{sc} \le 1 \qquad f_{wt} \le 1 \\
& f_{sd} \le 1 \qquad f_{xt} \le 1 \\
& f_{se} \le 1 \qquad f_{yt} \le 1 \\
& f_{sf} \le 1 \qquad f_{zt} \le 1 \\
& \text{all variables nonnegative.}
\end{aligned}
$$

An optimal solution to this LP is $v = 5$, $f_{aw} = f_{by} = f_{cx} = f_{dv} = f_{ez} = 1$, $f_{sa} = f_{sb} = f_{sc} = f_{sd} = f_{se} = f_{vt} = f_{wt} = f_{xt} = f_{yt} = f_{zt} = 1$, and all other variables 0. Therefore, a maximum matching in this graph is as shown below.



(This maximum matching is not unique.) □

6. A small trucking company has a fleet of five trucks, and on a certain day has seven loads to deliver. In the following tables, the capacities of the trucks and the sizes of the loads are both given in units of 1000 pounds.

Truck	Capacity	Daily cost
1	3	\$200
2	6	\$300
3	6	\$400
4	8	\$350
5	11	\$500

Load	Weight
A	1
B	2
C	3
D	4
E	4
F	5
G	8

The daily cost for a truck must be paid if that truck is to be used to make any deliveries. Because of their locations, loads A and D cannot be delivered by the same truck, nor can loads B and E. Formulate an integer program to determine which loads should be assigned to each truck in order to minimize the total daily cost.

▷ **Solution.** For $i \in \{\mathrm{A},\mathrm{B},\mathrm{C},\mathrm{D},\mathrm{E},\mathrm{F},\mathrm{G}\}$ and $j \in \{1,2,3,4,5\}$, let $x_{ij} \in \{0,1\}$ indicate whether load i is to be delivered by truck j. For $j \in \{1,2,3,4,5\}$, let $y_j \in \{0,1\}$ indicate whether the daily cost is to be paid for truck j.

Our objective is to minimize the total daily cost, which is $200y_1 + 300y_2 + 400y_3 + 350y_4 + 500y_5$.

We have several sets of constraints. First, every load must be delivered by exactly one truck, so

$$\sum_{j=1}^{5} x_{ij} = 1 \qquad \text{for all } i \in \{\mathrm{A},\mathrm{B},\mathrm{C},\mathrm{D},\mathrm{E},\mathrm{F},\mathrm{G}\}.$$

The truck capacities cannot be exceeded, so

$$x_{\mathrm{A}j} + 2x_{\mathrm{B}j} + 3x_{\mathrm{C}j} + 4x_{\mathrm{D}j} + 4x_{\mathrm{E}j} + 5x_{\mathrm{F}j} + 8x_{\mathrm{G}j} \le W_j \qquad \text{for all } j \in \{1,2,3,4,5\},$$

where W_j is the capacity of truck j. The daily cost must be paid to use each truck, so

$$x_{ij} \le y_j \qquad \text{for all } i \in \{\mathrm{A},\mathrm{B},\mathrm{C},\mathrm{D},\mathrm{E},\mathrm{F},\mathrm{G}\} \text{ and all } j \in \{1,2,3,4,5\},$$

because if $y_j = 0$ then x_{ij} must also be 0. Finally, loads A and D cannot be delivered by the same truck, nor can loads B and E, so

$$x_{\mathrm{A}j} + x_{\mathrm{D}j} \le 1 \qquad \text{for all } j \in \{1,2,3,4,5\},$$
$$x_{\mathrm{B}j} + x_{\mathrm{E}j} \le 1 \qquad \text{for all } j \in \{1,2,3,4,5\}.$$

Therefore we have the following integer program.

$$
\begin{aligned}
\text{minimize} \quad & 200y_1 + 300y_2 + 400y_3 + 350y_4 + 500y_5 \\
\text{subject to} \quad & x_{A1} + x_{A2} + x_{A3} + x_{A4} + x_{A5} = 1 \\
& x_{B1} + x_{B2} + x_{B3} + x_{B4} + x_{B5} = 1 \\
& x_{C1} + x_{C2} + x_{C3} + x_{C4} + x_{C5} = 1 \\
& x_{D1} + x_{D2} + x_{D3} + x_{D4} + x_{D5} = 1 \\
& x_{E1} + x_{E2} + x_{E3} + x_{E4} + x_{E5} = 1 \\
& x_{F1} + x_{F2} + x_{F3} + x_{F4} + x_{F5} = 1 \\
& x_{G1} + x_{G2} + x_{G3} + x_{G4} + x_{G5} = 1 \\
& x_{A1} + 2x_{B1} + 3x_{C1} + 4x_{D1} + 4x_{E1} + 5x_{F1} + 8x_{G1} \le 3 \\
& x_{A2} + 2x_{B2} + 3x_{C2} + 4x_{D2} + 4x_{E2} + 5x_{F2} + 8x_{G2} \le 6 \\
& x_{A3} + 2x_{B3} + 3x_{C3} + 4x_{D3} + 4x_{E3} + 5x_{F3} + 8x_{G3} \le 6 \\
& x_{A4} + 2x_{B4} + 3x_{C4} + 4x_{D4} + 4x_{E4} + 5x_{F4} + 8x_{G4} \le 8 \\
& x_{A5} + 2x_{B5} + 3x_{C5} + 4x_{D5} + 4x_{E5} + 5x_{F5} + 8x_{G5} \le 11 \\
& x_{A1} \le y_1, \quad x_{A2} \le y_2, \quad x_{A3} \le y_3, \quad x_{A4} \le y_4, \quad x_{A5} \le y_5 \\
& x_{B1} \le y_1, \quad x_{B2} \le y_2, \quad x_{B3} \le y_3, \quad x_{B4} \le y_4, \quad x_{B5} \le y_5 \\
& x_{C1} \le y_1, \quad x_{C2} \le y_2, \quad x_{C3} \le y_3, \quad x_{C4} \le y_4, \quad x_{C5} \le y_5 \\
& x_{D1} \le y_1, \quad x_{D2} \le y_2, \quad x_{D3} \le y_3, \quad x_{D4} \le y_4, \quad x_{D5} \le y_5 \\
& x_{E1} \le y_1, \quad x_{E2} \le y_2, \quad x_{E3} \le y_3, \quad x_{E4} \le y_4, \quad x_{E5} \le y_5 \\
& x_{F1} \le y_1, \quad x_{F2} \le y_2, \quad x_{F3} \le y_3, \quad x_{F4} \le y_4, \quad x_{F5} \le y_5 \\
& x_{G1} \le y_1, \quad x_{G2} \le y_2, \quad x_{G3} \le y_3, \quad x_{G4} \le y_4, \quad x_{G5} \le y_5 \\
& x_{A1} + x_{D1} \le 1, \qquad x_{B1} + x_{E1} \le 1 \\
& x_{A2} + x_{D2} \le 1, \qquad x_{B2} + x_{E2} \le 1 \\
& x_{A3} + x_{D3} \le 1, \qquad x_{B3} + x_{E3} \le 1 \\
& x_{A4} + x_{D4} \le 1, \qquad x_{B4} + x_{E4} \le 1 \\
& x_{A5} + x_{D5} \le 1, \qquad x_{B5} + x_{E5} \le 1 \\
& x_{ij} \in \{0,1\} \quad \text{for all } i \in \{\mathrm{A},\mathrm{B},\mathrm{C},\mathrm{D},\mathrm{E},\mathrm{F},\mathrm{G}\} \text{ and all } j \in \{1,2,3,4,5\} \\
& y_j \in \{0,1\} \quad \text{for all } j \in \{1,2,3,4,5\}.
\end{aligned}
$$

The formulation is all that is necessary for this problem. If this integer program is solved, it is found that there are eight optimal feasible solutions, each having total cost \$1350:

1	A, B	A, B	B	B	C	C	C	C
2	F	F	A, F	A, F	A, F	A, E	B, D	F
3	—	—	—	—	—	—	—	—
4	D, E	G	D, E	G	D, E	G	G	D, E
5	C, G	C, D, E	C, G	C, D, E	B, G	B, D, F	A, E, F	A, B, G

For instance, the first solution above has $x_{A1} = x_{B1} = x_{C5} = x_{D4} = x_{E4} = x_{F2} = x_{G5} = 1$, $y_1 = y_2 = y_4 = y_5 = 1$, and all other variables 0. □