Solutions to Chapter I

Based on *Computational Complexity* [4]

Ricardo Magal

02/02/2026

Contents

Problems **2**
Problem 1.4.2 2
Problem 1.4.3 2
Problem 1.4.4 3
Problem 1.4.5 3
Problem 1.4.9 4
Problem 1.4.10 5
Problem 1.4.12 7
Problem 1.4.15 8

Problems

Problem 1.4.2

Solution. Consider the standard reachability search from a start vertex 1. Maintain a set S of discovered vertices, initialized as $S = \{1\}$. Repeatedly add a vertex $v \notin S$ whenever there exists an edge $(u, v) \in E$ with $u \in S$. Stop when no such edge exists.

(a) We prove by induction on i that if v is the i-th vertex added to S, then there exists a directed path from 1 to v.

Base case. The first vertex added is 1, and the length-0 path suffices.

Inductive step. Assume the claim holds for the first $i - 1$ added vertices. Let v be the i-th vertex added. By the rule of the algorithm, v is added because there exists an edge $(u, v) \in E$ with $u \in S$ at that moment. Since u was added earlier, by the induction hypothesis there is a path from 1 to u. Appending the edge (u, v) yields a path from 1 to v.

(b) We prove by induction on ℓ that if v is reachable from 1 by a directed path of ℓ edges, then the algorithm will add v to S.

Base case. If $\ell = 0$, then $v = 1$, and $1 \in S$ initially.

Inductive step. Assume the statement holds for all vertices reachable by a path of length $\ell - 1$, for some $\ell \geq 1$. Let v be reachable from 1 by a path of length ℓ, and fix such a path. Let u be the vertex immediately preceding v on this path. Then u is reachable by a path of length $\ell - 1$, hence u is added to S by the induction hypothesis. Once $u \in S$, the edge (u, v) witnesses that v is eligible to be added, so the algorithm will eventually add v (it continues until no eligible edge remains).

Parts (a) and (b) together imply that the algorithm adds exactly the vertices reachable from 1.

Problem 1.4.3

Solution. Implement the search using adjacency lists. For each vertex u, store its outgoing neighbors $\Gamma^+(u) = \{v : (u, v) \in E\}$. Maintain a boolean array $\texttt{seen}[\cdot]$ and a worklist (queue for breadth-first search (BFS) or stack for depth-first search (DFS)).

Initialize $\texttt{seen}[1] = \texttt{true}$ and push 1 into the worklist. While the worklist is nonempty, pop a vertex u and scan its adjacency list $\Gamma^+(u)$. For each neighbor $v \in \Gamma^+(u)$, the edge (u, v) is *processed* by performing the constant-time check of $\texttt{seen}[v]$ and, if false, setting it to true and pushing v into the worklist.

Each edge is processed at most once. A vertex u is popped at most once: after $\texttt{seen}[u]$ becomes true, u is inserted into the worklist at most once, and therefore its adjacency list is scanned at most once. Since an edge (u, v) appears only in the adjacency list of its tail u, it is examined only when u is popped. Hence each edge $(u, v) \in E$ is processed at most once.

Running time. The total cost of scanning all adjacency lists is

$$\sum_{u \in V} \Gamma^+(u) = E,$$

and the per-edge work is constant. Therefore the algorithm runs in (E) edge-processing steps (and in $(V+E)$ total time if one also counts initialization and worklist operations).

Problem 1.4.4

Solution. Let $G=(V,E)$ be a finite directed graph. Recall that G is directed acyclic graph (DAG) if it has no directed cycle.

(a) **Every DAG has a source.** Assume G is acyclic. We show that G has a vertex of indegree 0.

Suppose, for contradiction, that every vertex has indegree at least 1. Pick any $v_0 \in V$. Since v_0 has an incoming edge, choose v_1 with $(v_1, v_0) \in E$. Similarly, choose v_2 with $(v_2, v_1) \in E$, and continue, obtaining a sequence $v_0, v_1, v_2, \ldots$. Because V is finite, some vertex repeats: $v_i = v_j$ for some $i < j$. Then $(v_{i+1}, v_i), (v_{i+2}, v_{i+1}), \ldots, (v_j, v_{j-1})$ forms a directed cycle, contradicting acyclicity. Hence some vertex must have indegree 0, i.e., a source exists.

(b) **Acyclicity iff a topological ordering exists.**

($\Rightarrow$) Assume G is acyclic. Repeatedly select a source (guaranteed by (a)), assign it the next label $1, 2, \ldots$, and delete it together with its outgoing edges. Deleting a vertex cannot create a cycle, so the remaining graph stays acyclic and the process continues until all vertices are labeled. If a directed edge (u,v) remains when u is labeled, then v is still present; thus v is labeled later. Therefore every edge goes from a lower number to a higher number, i.e., the labeling is a topological ordering.

($\Leftarrow$) Conversely, assume the vertices can be numbered $1, \ldots, n$ so that every edge $(u,v) \in E$ satisfies $\mathrm{num}(u) < \mathrm{num}(v)$. Along any directed walk, the numbers strictly increase, so no directed cycle can exist. Hence G is acyclic.

(c) **Algorithm in $O(|E|+|V|)$ time, with constant work per edge.** Implement the idea in (b) via Kahn's algorithm.

1. Compute $\mathrm{indeg}[v]$ for all $v \in V$ by scanning the edge list once and incrementing $\mathrm{indeg}[v]$ for each edge $(u,v) \in E$.
2. Initialize a queue (a worklist) with all vertices v such that $\mathrm{indeg}[v] = 0$.
3. Repeatedly pop a vertex u from the queue, output it next in the ordering, and for each outgoing edge (u,w), decrement $\mathrm{indeg}[w]$. If $\mathrm{indeg}[w]$ becomes 0, push w into the queue.

If the algorithm outputs all vertices, then it has produced a topological ordering, so G is acyclic by (b). If it stops early (queue empty while some vertices remain), then the remaining subgraph has no source, so by (a) it must contain a directed cycle; hence G is not acyclic.

Each edge (u,w) is examined exactly once when u is popped, and causes one constant-time decrement, so the total edge-processing work is $O(|E|)$. The remaining work (initialization, queue operations) is $O(|V|)$. Thus the algorithm runs in $O(|E|+|V|)$ time.

Problem 1.4.5

Solution. We consider an undirected graph $G=(V,E)$. A bipartite graph is a graph whose vertices can be partitioned into two sets A, B such that every edge has one endpoint in A and the other in B.

(a) **G is bipartite iff it has no odd cycle.**

($\Rightarrow$) Suppose G is bipartite with partition $V = A \cup B$. Along any cycle, vertices must alternate between A and B (because every edge crosses the cut (A, B)). Therefore the cycle length is even. Hence there is no odd cycle.

($\Leftarrow$) Suppose G has no odd cycle. Fix a connected component and choose a root r. Define a two-coloring by BFS layers: place a vertex v in A iff the shortest-path distance (r, v) is even, and in B iff (r, v) is odd.

We claim that no edge has both endpoints in A (and similarly for B). If there were an edge $\{x, y\} \in E$ with $(r, x) \equiv (r, y) \pmod 2$, let P_x and P_y be shortest paths from r to x and r to y. Let z be their last common vertex (the point where they diverge). Then the subpaths from z to x and from z to y, together with the edge $\{x, y\}$, form a cycle of length $(z, x) + (z, y) + 1$. Since $(r, x) \equiv (r, y) \pmod 2$, we have $(z, x) \equiv (z, y) \pmod 2$, so $(z, x) + (z, y) + 1$ is odd. This produces a odd cycle, contradiction. Therefore every edge crosses between A and B, and the component is bipartite. Doing this independently for each component yields a bipartition of G.

(b) **Polynomial-time algorithm to test bipartiteness.** Run BFS (or DFS) in each connected component while maintaining a two-coloring. Initialize all vertices as uncolored. For each uncolored start vertex r, color it 0 and perform BFS. When exploring an edge $\{u, v\}$:

- if v is uncolored, assign it color $1 - \text{color}(u)$ and continue;
- if v is already colored and $\text{color}(v) = \text{color}(u)$, reject (an odd cycle exists).

If all edges are processed without conflict, accept and return the induced partition.

With adjacency lists, each edge is examined at most twice (once from each endpoint), so the running time is $O(|V| + |E|)$.

Problem 1.4.9

Solution. Let $p(n)$ be a polynomial and let $c > 0$ be a constant.

General statement. Since p is a polynomial, there exist constants $a > 0$ and $k \in$ such that $p(n) \leq an^k$ for all $n \geq 1$. We want $2^{cn} > p(n)$, and it suffices to ensure

$$2^{cn} > an^k.$$

Taking of both sides yields the equivalent inequality

$$cn > a + kn.$$

The left-hand side grows linearly in n, while the right-hand side grows on the order of $\log n$. Hence there exists n_0 such that for all $n \geq n_0$ the inequality holds, proving the claim.

(a) Here $p(n) = n^2$ and $c = 1$. We need $2^n > n^2$ for all $n \geq n_0$.

Direct check gives $2^4 = 16 = 4^2$ and $2^5 = 32 > 25$. For $n \geq 5$, the function

$$g(n) = \frac{2^n}{n^2}$$

is strictly increasing, since

$$\frac{g(n+1)}{g(n)} = \frac{2^{n+1}/(n+1)^2}{2^n/n^2} = 2\left(\frac{n}{n+1}\right)^2 > 1.$$

Therefore $2^n/n^2 \ge 2^5/5^2 > 1$ for all $n \ge 5$, so $n_0 = 5$ works.

(b) Here $p(n) = 100n^{100}$ and $c = \frac{1}{100}$. We need

$$2^{n/100} > 100n^{100}.$$

Taking yields

$$\frac{n}{100} > 100 + 100n.$$

Define

$$\phi(n) = \frac{n}{100} - 100 - 100n.$$

For $n > 0$,

$$\phi'(n) = \frac{1}{100} - \frac{100}{n\ln 2}.$$

Thus $\phi'(n) > 0$ for all $n \ge 15000$, because then $\frac{100}{n\ln 2} \le \frac{100}{15000\ln 2} < \frac{1}{100}$. So ϕ is increasing for $n \ge 15000$. It suffices to find one $n \ge 15000$ with $\phi(n) > 0$.

Take $n_0 = 200000$. Since $(200000) < 18$ and $100 < 7$, we have

$$100 + 100(200000) < 7 + 1800 = 1807 < 2000 = \frac{200000}{100}.$$

Hence $\phi(200000) > 0$, and by monotonicity $\phi(n) > 0$ for all $n \ge 200000$. Therefore $n_0 = 200000$ works.

Problem 1.4.10

Solution. Let the functions be:

$$(a)\ n^2, \quad (b)\ n^3, \quad (c)\ n^2\log n, \quad (d)\ 2^n,$$

$$(e)\ n^n, \quad (f)\ n^{10\log_2 n}, \quad (g)\ 2^{2n}, \quad (h)\ 2^{2n+1}, \quad (j)\ \begin{cases} n^2 & \text{if } n \text{ is odd,} \\ 2^n & \text{if } n \text{ is even.} \end{cases}$$

We use standard $O(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$. We also write $f \in o(g)$ (see $o(\cdot)$) to mean $\lim_{n\to\infty} f(n)/g(n) = 0$.

Step 1: A strict growth chain for (a),(b),(c),(d),(e),(f),(g),(h). We claim the following asymptotic hierarchy:

$$n^2 \in o(n^2\log n) \in o(n^3) \in o\left(n^{10\log_2 n}\right) \in o(2^n) \in o(2^{2n}), \quad 2^{2n} \in \Theta(2^{2n+1}), \quad 2^{2n+1} \in o(n^n).$$

Each step is justified as follows:

- $\dfrac{n^2}{n^2\log n} = \dfrac{1}{\log n} \to 0.$
- $\dfrac{n^2\log n}{n^3} = \dfrac{\log n}{n} \to 0.$

- $\dfrac{n^3}{n^{10\log_2 n}} = n^{3-10\log_2 n} \to 0$ since $10\log_2 n \to \infty$.
- Compare $n^{10\log_2 n}$ to 2^n by taking $\log_2$:
$$\log_2\left(n^{10\log_2 n}\right) = 10(\log_2 n)^2 = o(n) = \log_2(2^n),$$
hence $n^{10\log_2 n} \in o(2^n)$.
- $\dfrac{2^n}{2^{2n}} = 2^{-n} \to 0$.
- $2^{2n+1} = 2 \cdot 2^{2n}$, so $2^{2n} \in \Theta(2^{2n+1})$.
- Compare 2^{2n+1} to n^n by taking $\log_2$:
$$\log_2(2^{2n+1}) = 2n+1, \qquad \log_2(n^n) = n\log_2 n,$$
and $n\log_2 n \gg 2n$, so $2^{2n+1} \in o(n^n)$.

Consequence (pairs not involving (j)). For any two distinct functions among $(a), (b), (c), (d), (e), (f), (g), (h)$, the slower-growing one is in $O(\cdot)$ of the faster-growing one, and the faster-growing one is in $\Omega(\cdot)$ of the slower. The only $\Theta(\cdot)$ equivalence among these is (g) and (h), since they differ by a factor 2.

Step 2: The exceptional function (j). Define
$$J(n) = \begin{cases} n^2 & \text{if } n \text{ is odd,} \\ 2^n & \text{if } n \text{ is even.} \end{cases}$$
Then $J(n) \geq n^2$ for all n, and $J(2k) = 2^{2k}$.

- Versus (a) n^2: $J \in \Omega(n^2)$ (trivial) but $J \notin O(n^2)$ because on even n, $J(n) = 2^n$ and $2^n/n^2 \to \infty$. Thus J and n^2 are not $\Theta(\cdot)$-equivalent. Equivalently, $n^2 \in O(J)$ and $J \in \Omega(n^2)$.
- Versus (c) $n^2\log n$, (b) n^3, and (f) $n^{10\log_2 n}$: J is asymptotically incomparable with each of these functions. Indeed, on even n, $J(n) = 2^n$ dominates any of them; hence $J \notin O(g)$. On odd n, $J(n) = n^2$ is dominated by each of them for large n; hence $J \notin \Omega(g)$. Therefore neither (i) nor (ii) holds (and thus not (iii)) for these pairs.
- Versus (d) 2^n: $J \in O(2^n)$ because for even n, $J(n) = 2^n$, and for odd $n \geq 5$, $n^2 \leq 2^n$. But $J \notin \Omega(2^n)$ since on odd n, $J(n)/2^n = n^2/2^n \to 0$. So J is strictly smaller than 2^n along an infinite subsequence. Also $2^n \notin O(J)$, because on odd n, $2^n/J(n) = 2^n/n^2 \to \infty$.
- Versus (g) 2^{2n} and (h) 2^{2n+1}: $J \in O(2^{2n})$ (and hence $O(2^{2n+1})$), but $J \notin \Omega(2^{2n})$ since on odd n, $J(n) = n^2$ and $n^2/2^{2n} \to 0$. Likewise, $2^{2n} \notin O(J)$ because on even n, $2^{2n}/J(n) = 2^{2n}/2^n = 2^n \to \infty$.
- Versus (e) n^n: $J \in O(n^n)$ (since $J(n) \leq 2^n \leq n^n$ for all $n \geq 2$), but $J \notin \Omega(n^n)$ because on odd n, $J(n) = n^2$ and $n^2/n^n = n^{2-n} \to 0$. Also $n^n \notin O(J)$ since on even n, $n^n/J(n) = n^n/2^n \to \infty$.

This fully determines which of (i) $f \in O(g)$, (ii) $f \in \Omega(g)$, or (iii) $f \in \Theta(g)$ holds for any pair from the list: use the total order in Step 1 for pairs not involving (j), and the case analysis above for pairs involving J.

Problem 1.4.12

Solution. We analyze the augmentation algorithm for maximum flow (Ford–Fulkerson), under the rule: at each iteration, choose an augmenting path of minimum number of edges in the current residual network $N(f)$, and augment by the bottleneck residual capacity along that path. This is the Edmonds–Karp algorithm strategy. [1, 2]

For a flow f, let $_f(v)$ be the length of a shortest directed path from s to v in $N(f)$ (or ∞ if none exists). Let f' be the flow obtained after one augmentation.

(a) **Distances do not decrease.** We claim that for every vertex $v \in V$, $_{f'}(v) \geq_f (v)$.

Consider a shortest s-to-v path Q in $N(f')$. Let (x, y) be the last edge of Q, so $y = v$ and $_{f'}(x) =_{f'} (y) - 1$. Choose v with $_{f'}(v) <_f (v)$ minimizing $_{f'}(v)$. Then $_{f'}(x) \geq_f (x)$ by minimality.

If (x, y) is also an edge of $N(f)$, then

$$_f(y) \leq_f (x) + 1 \leq_{f'} (x) + 1 =_{f'} (y),$$

contradicting $_{f'}(y) <_f (y)$.

Otherwise, (x, y) was not residual under f but becomes residual under f'. The only new residual edges created by an augmentation are reverse edges of the augmented path P: thus (x, y) must be of the form $(x, y) = (j, i)$ where (i, j) lies on P in the forward direction. Since P is a shortest augmenting path in $N(f)$, we have $_f(j) =_f (i) + 1$. Also $_{f'}(x) =_{f'} (j) \geq_f (j)$ by minimality, so

$$_{f'}(i) =_{f'} (j) + 1 \geq_f (j) + 1 =_f (i) + 2,$$

and hence $_{f'}(y) =_{f'} (i) \geq_f (i) + 2$. In particular $_{f'}(y) \geq_f (y)$, contradiction. Therefore distances cannot decrease.

(b) **Bottleneck reversal forces a distance increase.** Suppose that at some stage, on the chosen shortest augmenting path P in $N(f)$, the edge (i, j) is a bottleneck edge, i.e., it attains the minimum residual capacity along P. Then (i, j) becomes saturated after augmentation and the reverse residual edge (j, i) appears in $N(f')$.

Assume that at a later stage, (j, i) becomes a bottleneck edge on some chosen shortest augmenting path in $N(g)$, for a later flow g. Since (j, i) lies on a shortest path in $N(g)$, we have $_g(i) =_g (j) + 1$.

From the stage when (i, j) was on a shortest path, we know $_f(j) =_f (i) + 1$. By part (a), distances never decrease across augmentations, so $_g(j) \geq_f (j) =_f (i) + 1$. Therefore

$$_g(i) =_g (j) + 1 \geq (_f(i) + 1) + 1 =_f (i) + 2.$$

In particular, the distance from s to i must increase before the reverse edge (j, i) can be a bottleneck.

(c) **At most $O(|E||V|)$ augmentations.** Each augmentation saturates at least one bottleneck residual edge on the chosen shortest augmenting path. Fix an original directed edge $(u, v) \in E$. Consider the events where either (u, v) or (v, u) becomes a bottleneck edge of a chosen shortest augmenting path (these are the only times this original edge contributes a bottleneck in the residual network).

By part (b), once (u, v) is a bottleneck, before (v, u) can later be a bottleneck, the distance (u) must increase by at least 2. Distances are always at most $|V| - 1$ when finite, and by part (a) they never decrease. Hence the number of alternations of bottleneck direction for this pair is $O(|V|)$, and therefore each original edge can be a bottleneck at most $O(|V|)$ times overall.

Since each augmentation saturates at least one bottleneck edge, the total number of augmentations is at most $O(|E||V|)$.

Problem 1.4.15

Solution. We are given a complete weighted graph on cities $\{1, \ldots, n\}$ with distances d_{ij}. For each subset $S \subseteq \{2, \ldots, n\}$ and each $j \in S$, define $c[S, j]$ as the length of the shortest walk that starts at city 1, visits every city in S exactly once, and ends at city j. This is the classical dynamic program for Traveling Salesman Problem. [3]

(a) **Dynamic programming recurrence and algorithm.** The base cases are

$$c[\{j\}, j] = d_{1j} \qquad \text{for all } j \in \{2, \ldots, n\}.$$

For $|S| \geq 2$, the recurrence is

$$c[S, j] = \min_{i \in S \setminus \{j\}} \left(c[S \setminus \{j\}, i] + d_{ij} \right).$$

Compute the values in increasing order of $|S|$ (from 1 to $n - 1$). Concretely:

- For $k = 1, 2, \ldots, n - 1$:
  - For each subset $S \subseteq \{2, \ldots, n\}$ with $|S| = k$:
    - For each $j \in S$: set $c[S, j]$ by the recurrence above.

Each state (S, j) depends only on strictly smaller subsets, so the ordering by $|S|$ is valid.

(b) **Running time and space.** There are $\sum_{k=1}^{n-1} \binom{n-1}{k} \cdot k = (n-1)2^{n-2} = \Theta(n2^n)$ states (S, j). For each state, the recurrence takes a minimum over at most $|S| - 1 \leq n$ choices of i. Therefore the total running time is

$$\Theta(n2^n) \cdot \Theta(n) = \Theta(n^2 2^n),$$

which is $O(n^2 2^n)$ as required.

Space is $\Theta(n2^n)$ to store all values $c[S, j]$. If we also store an argmin predecessor pointer for each state (to reconstruct an optimal tour), the space remains $\Theta(n2^n)$.

(c) **Shortest path from 1 to n without visiting all cities.** If we only want the shortest path from 1 to n (in the usual graph-theoretic sense), tracking the exact subset S of visited cities is unnecessary: shortest paths with nonnegative weights can be assumed simple path (no repeated vertices), hence they use at most $n - 1$ edges.

A standard polynomial-time dynamic program indexed only by the number of edges suffices (this is the Bellman–Ford style DP). Define $D[k, v]$ as the minimum weight of a path from 1 to v using at most k edges. Initialize $D[0, 1] = 0$ and $D[0, v] = \infty$ for $v \neq 1$. For $k \geq 1$,

$$D[k, v] = \min\Big(D[k-1, v], \; \min_{u \neq v} \big(D[k-1, u] + d_{uv} \big) \Big).$$

After $n-1$ iterations, $D[n-1, n]$ equals the shortest-path distance from 1 to n.

This runs in $O(n)$ iterations; each iteration computes n values, each via a minimum over $O(n)$ predecessors, so the time is $O(n^3)$ and the space is $O(n^2)$. Thus the problem is solvable in polynomial time, and the dependence on $|S|$ can be replaced by the scalar parameter k (number of edges).

Acronyms

BFS breadth-first search. 2, 4

DAG directed acyclic graph. 3

DFS depth-first search. 2, 4

Glossary

$\Omega(\cdot)$ Big-Omega notation: $f \in \Omega(g)$ if there exist constants $c > 0$ and n_0 such that $f(n) \geq c\,g(n)$ for all $n \geq n_0$. 5, 6

$\Theta(\cdot)$ Big-Theta notation: $f \in \Theta(g)$ if $f \in O(g)$ and $f \in \Omega(g)$. 5, 6

adjacency list A representation of a graph where each vertex u stores the list of its outgoing (or incident) neighbors; scanning all lists processes each edge once. 2, 4

argmin For a function f on a finite domain, an *argmin* is an input where f attains its minimum value. 8

asymptotically incomparable Two functions f, g are asymptotically incomparable if neither $f \in O(g)$ nor $f \in \Omega(g)$ holds. 6

augmenting path An s-to-t path in the residual network along which the flow value can be increased by augmenting. 7

bipartite graph A graph whose vertices can be partitioned into two sets A, B such that every edge has one endpoint in A and the other in B. 3

bottleneck edge On an augmenting path, an edge whose residual capacity is minimum along the path; it limits the augmentation amount. 7

Edmonds–Karp algorithm A specialization of Ford–Fulkerson that always augments along a shortest (fewest-edge) augmenting path; it runs in $O(|V||E|^2)$ time. 7

indegree For a directed graph, the number of edges entering a vertex v. 3

Kahn's algorithm A linear-time algorithm for topological sorting that repeatedly removes sources, updating indegrees of out-neighbors. 3

maximum flow The problem of sending the largest possible amount of flow from a source s to a sink t in a capacitated directed network, subject to capacity constraints and flow conservation. 7

$O(\cdot)$ Big-O notation: $f \in O(g)$ if there exist constants $c > 0$ and n_0 such that $f(n) \leq c\,g(n)$ for all $n \geq n_0$. 5, 6

$o(\cdot)$ Little-o notation: $f \in o(g)$ if $\lim_{n\to\infty} f(n)/g(n) = 0$. 5

odd cycle A cycle whose number of edges is odd. 4

residual capacity The additional amount of flow that can be sent along a residual edge; for an edge (u, v) it is $c(u, v) - f(u, v)$ in the forward direction and $f(u, v)$ in the reverse direction. 7

residual network Given a flow f, the directed network $N(f)$ whose edges indicate how the current flow can be increased or rerouted via residual capacities. 7

simple path A path that does not repeat vertices. 8

source A vertex of indegree 0 in a directed graph. 3

topological ordering A numbering of the vertices $1, \ldots, n$ such that every edge goes from a lower number to a higher number. 3

Traveling Salesman Problem Given distances between n cities, find a minimum-length tour that starts at a city, visits every city exactly once, and returns to the start. 8

two-coloring An assignment of two colors (or labels $0, 1$) to vertices so that adjacent vertices receive different colors. 4

worklist A data structure (typically a queue or stack) that stores discovered but not-yet-processed vertices during graph search. 2, 3

References

[1] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

[2] L. R. Ford and D. R. Fulkerson. *Flows in Networks.* Princeton University Press, 1962.

[3] Michael Held and Richard M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.

[4] Christos H. Papadimitriou. *Computational Complexity.* Addison–Wesley, 1994.