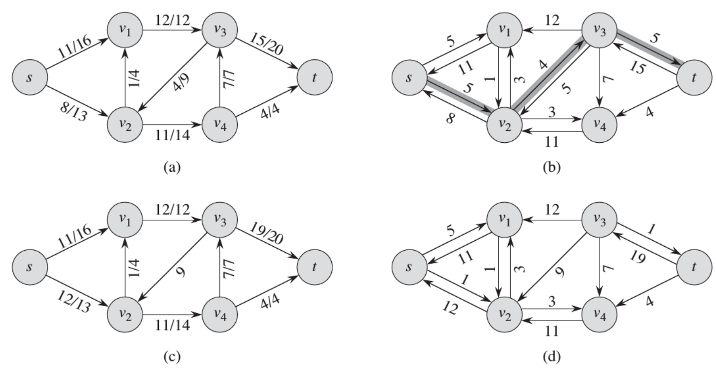
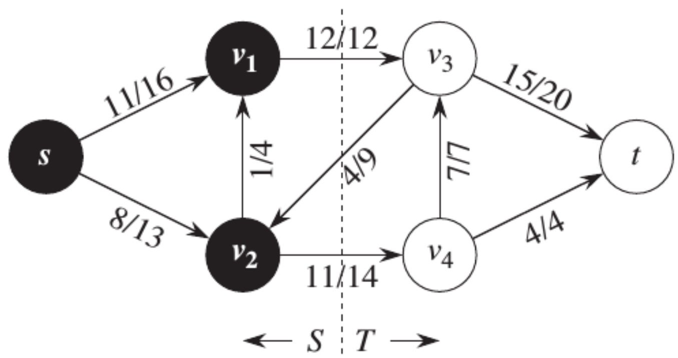
Maximum Flow



(a) (b) (c) (d)



A **flow network** $G = (V, E)$ is a directed graph.

1) Each edge has **capacity** $c(u, v) \geq 0$.
2) Two special vertices: **source** s and **sink** t.

A **flow** is a function $f : V \times V \to \mathbb{R}$ that has two properties:

1) **Capacity constraint**: For all $u, v \in V$

$$0 \leq f(u, v) \leq c(u, v)$$

2) **Flow constraint**: For all $u \in V - \{s, t\}$

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$$

Maximum flow is a flow f with maximum **value**

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s).$$

Residual network $G_f = (V, E_f)$ where

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$$

and **residual capacity**

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & (u, v) \in E, \\ f(v, u) & (v, u) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

An **augmenting path** p is a simple path from s to t in the residual network G_f. The residual capacity of path p is

$$c_f(p) = \min\{c_f(u, v) : (u, v) \text{ is on } p\}.$$

A **cut** (S, T) of flow network G is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$.

A **minimum cut** of a network is a cut whose **capacity**

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$$

is minimum over all cuts of the network.

Max-flow Min-cut Theorem

1) f is a maximum flow in G.
2) The residul network G_f contains no augmenting paths.
3) $|f| = c(S, T)$ for some cut (S, T) of G.

Ford-Fulkerson(G, s, t)

1) Set $f(u, v) = 0$ for all $(u, v) \in E$
2) **while** there exists a path p from s to t in the residual network G_f
3) ____ **for** each edge $(u, v) \in p$
4) ____ ____ **if** $(u, v) \in E$
5) ____ ____ ____ $f(u, v) = f(u, v) + c_f(p)$
6) ____ ____ **else** $f(v, u) = f(v, u) - c_f(p)$

Dynamic Programming

Steps for developing dynamic programming algorithm:

1) Characterize the optimal substructure.
2) Recursively define the value of an optimal solution.
3) Compute the value of the optimal solution, typically in a bottom-up fashion.
4) Construct an optimal solution from computed information.

Elements of dynamic programming:

1) **Optimal substructure**: An optimal solution contains within it optimal solutions to subproblems.
2) **Overlapping subproblems**: The recursive algorithm revisits the same problems repeatedly.

Subproblem graph: Embodies information on how subproblems depend on one another.

Longest increasing subsequence:

1) $L(j)$ is the distance of the longest increasing subsequence upto j
2) $L(j) = 1 + \max\{L(i) : (i, j) \in E\}$

Knapsack

1) $K(w, j)$ is maximum values achievable using a knapsack of capacity w and items $1, ..., j$.
2) $K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$

Independent sets on trees

1) $I(u)$ is the size of largest independent set of subtree handing from u
2) $I(u) = \max\left\{1 + \sum_{\text{grandchildren } w \text{ of } u} I(w), \sum_{\text{children } w \text{ of } u} I(w)\right\}$

Linear Programming

Standard form: Maximize the **value** of the **objective function** given set of **constraints**

$$
\begin{aligned}
\text{maximize} \quad & \mathbf{c}^T \mathbf{x} \\
\text{subject to} \quad & A\mathbf{x} \leq \mathbf{b} \\
& \mathbf{x} \geq 0
\end{aligned}
$$

- **Feasible solution**: The variables $\mathbf{x}$ that satisfy the constraints
- **Feasible region**: convex region consisting formed by the set of feasible solutions
- **Integer linear programming**: The values $\mathbf{x}$ are constrained to integer values $\mathbb{N}$. NP-hard!
- **Simplex algorithm**: principle?

Single source shortest path: Given weighted directed graph $G = (V, E)$, with weight function $w : E \to \mathbb{R}$, a source vertex s, and destionation vertex t.

$$
\begin{aligned}
\text{maximize} \quad & d_t \\
\text{subject to} \quad & d_v \leq d_u + w(u, v) \text{ for each edge } (u, v) \in E \\
& d_s = 0.
\end{aligned}
$$

Maximum flow can be formulated as a linear program:

$$
\begin{aligned}
\text{maximize} \quad & \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) \\
\text{subject to} \quad & f_{uv} \leq c(u, v) && \text{for each edge } u, v \in V, \\
& \sum_{v \in V} f_{vu} = \sum_{v \in V} f_{uv}, && \text{for each } u \in V - \{s, t\} \\
& f_{uv} \geq 0 && \text{for each edge } u, v \in V.
\end{aligned}
$$

NP and Reductions

- **NP** class of all search problems.
- **P** class of all search problems that can be solved in polynomial time.
- **P≠NP** Are there search problems that cannot be solved in polynomial time?
- A search problem is **NP-complete** if all other search problems reduce to it.
- All problems in **NP** reduce to CIRCUIT SAT which reduces to SAT.

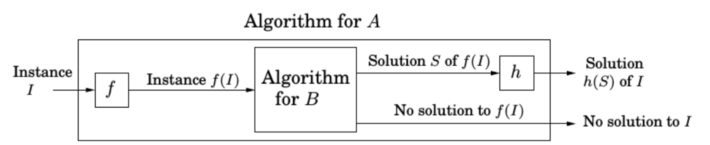
Satisfiability (SAT) is a problem of finding a **satisfying truth assignment** to a boolean formula. For boolean formula in conjunctive normal form (CNF)

$$(x \vee y \vee z) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg x) \wedge (\neg x \vee \neg y \vee \neg z)$$

a satisfying assignment is an assignment of `false` or `true` to each variable so that every clause contains a literal whose value is `true`.

A **search problem** is specified by an algorithm $\mathcal{C}$ that takes two inputs, an instance I and a proposed solution S, and runs in time polynomial in $|I|$. We say S is a solution to I if and only if $\mathcal{C}(I, S) = true$.

A **reduction** from search problem A to search problem B:



1) f is a *polynomial-time algorithm* that transforms any instance I of A into an instance of B.
2) h is a *polynomial-time algorithm* that maps any solution S of $f(I)$ of back into a solution $h(S)$ of I.