AS.601.433 Intro to Algorithms

Author: Tian Zhou

Institute: Johns Hopkins University

Date: December 14, 2023

Version: 1

Year: Fall 2023

***Textbook**: Algorithm Design - Kleinberg and Tardos*

Contents

Chapter 1 Stable Matching

1.1 Stable Matching Problem

1.1.1 Stable Matching Problem

Input: A set A of n objects and a set B of n objects, along with a preference list of A and B.

Perfect Matching: A matching M is a set of ordered pairs in $A \times B$ such that only one pair in M contains a for all $a \in A$ and the similar statement holds for $b \in B$. The matching M is perfect if M is a bijective function, namely $|M| = n$.

Definition 1.1 (Unstable Pair)

Given a perfect matching, a and b form an unstable pair if both a prefers b than $M(a)$ and b prefers a than $M(b)$. In other words, An unstable pair $a - b$ could each improve by joint action. ♣

Stable Matching Problem: Given the preference lists, find a stable matching (if one exists).

1.1.2 Gale-Shapley Algorithm

Algorithm 1 Gale-Shapley

```
Initialize M to empty matching
while Some element $a \in A$ is unmatched and hasn't proposed to every $b \in B$ do
    $b \leftarrow$ first $a$ on $b$'s list to which $a$ has not yet proposed.
    if $b$ is unmatched then
        Add $(a, b)$ to matching $M$.
    else if $b$ prefers $a$ to current partner $a'$ then
        Replace $(a', b)$ with $(a, b)$ in matching $M$.
    else
        $b$ rejects $a$.
    end if
end while
return stable matching $M$.
```

Property *(1) Element $a \in A$ propose to $b \in B$ in decreasing order of preference.*
(2) Once $b \in B$ is matched, then b never becomes unmatched, namely only trades up are possible.

Property *Algorithm terminates after at most n^2 iteration of while loop.*

Proof Each proposal through the while loop must be different from previous proposals, and there are n^2 distinct proposals, so the while loop terminates after at most n^2 iterations. ■

Property *Gale-Shapley outputs a matching.*

Proof For all $a \in A$, a proposes only if a unmatched, so a matches to at most one $b \in B$. For all $b \in B$, if b accepts the proposal of a, whether b is unmatched or b then rejects the previous proposal from a', so b matches to at most one $a \in A$. ■

Property *In Gale-Shapley matching, every element in A and every elements in B get matched.*

Proof The loop terminates only if all $a \in A$ is matched or every $b \in B$ is proposed by some unmatched $a \in A$. For the latter case, there exists $b \in B$ unmatched, so a never proposes to b by the second property, contradiction. For the first case, since $|A| = |B|$ and M is a matching, every element in B get matched. ■

Property *In Gale–Shapley matching M^*, there are no unstable pairs.*

Proof Consider $(a, b) \notin M^*$. Case 1: if a never proposed to a, then a prefers $M^*(a)$ than b since a proposes in a decreasing order, it follows that (a, b) is not unstable. Case 2: if a proposed to b, b rejects or renounces to $M^*(b)$, so b prefers $M^*(b)$ over a, implying that (a, b) is not unstable. ■

Corollary 1.1

The Gale–Shapley algorithm guarantees to find a stable matching for any problem instance within $O(n^2)$. ♡

1.1.3 A-Optimality

Definition 1.2 (Valid Partner)

*Element $b \in B$ is a **valid partner** for element $a \in A$ if there exists any stable matching in which a and b are matched.* ♣

Proposition 1.1

Gale-Shapley matching M^* is A-optional (namely, each $a \in A$ receives best valid partner). ♠

Proof Proof by contradiction. Assume a is the first element in A be rejected from its best valid partner b, and b forms commitment to a'. Then there exists b' such that $(a, b), (a', b') \in M$ for some stable matching M. Since b rejected a, we have $a' >_b a$. Since a' was not rejected by b' upon rejected by b, so a' has not yet proposed to b', implying that $b >_{a'} b'$. Therefore, (a', b) is an unstable pair in M, contradicting the fact that M is stable. ■

Proposition 1.2

Gale-Shapley matching M^* is B-pessimal (namely, each $b \in B$ receives worst valid partner). ♠

Proof Proof by contradiction. Let a denotes the worst valid partner of b, and suppose $(a', b), (a, b') \in M$ where

$a \neq a'$, $b \neq b'$. It is obvious that $a' >_b a$, and by A-optimality, $b >_{a'} b'$. Thus (a, b) is an unstable pair, resulting in a contradiction. ■

Chapter 2 Basics of Algorithm Analysis

2.1 Asymptotic Order of Growth

2.1.1 Computational Tractability

Efficient Algorithm An algorithm is *poly-time* (Polynomial running time) if there exists constants $a > 0$ and $b > 0$ such that for every input size of n, the complexity is no larger than an^b primitive computational steps. We say that an algorithm is *efficient* if it has a polynomial running time.

Worst-case Analysis The worst case analysis yields the running time guarantee for any input size n. It generally captures efficiency and hard to find effective alternative. However, there are some exponential-time algorithms are used widely in practice because the worst case instances don't arise.

Other Types fo Analyses *Probabilistic*: expected running time of a randomized algorithm. Amortized: worst-case running time for any sequence of n operations. Also, average case analysis, smoothed analysis, etc. are in used.

2.1.2 Asymptotic Order of Growth

Definition 2.1 (Upper Bound, Big O Notation $O(f)$)

$f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $N \geq 0$ such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq N$. ♣

Note: (a) We sometimes use $f(n) \in O(g(n))$ to denote $f(n) \in O(g(n))$.

(b) We sometimes extend the domain and the codomain of $f : \mathbb{N} \to \mathbb{N}$ to real.

Property *The Big O Notation has the following properties:*

1. *Reflexivity: $f \in O(f)$;*
2. *Transitivity: if $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$.*
3. *Constants: if $f \in O(g)$ and $c > 0$, then $cf \in O(g)$;*
4. *Products: if $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$, then $f_1 f_2 \in O(g_1 g_2)$;*
5. *Sums: if $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$, then $f_1 + f_2 \in O(\max\{g_1, g_2\})$;*

Definition 2.2 (Lower Bound, Big Omega Notation $\Omega(f)$)

$f(n)$ is $\Omega(g(n))$ if there exist constants $c > 0$ and $N \geq 0$ such that $0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq N$. ♣

Definition 2.3 (Tight Bound, Big Theta Notation $\Theta(f)$)

$f(n)$ is $\Theta(g(n))$ if there exist constants $c_1, c_2 > 0$ and $N \geq 0$ such that $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq N$. ♣

Proposition 2.1

If $\lim_{n\to\infty} f(n)/g(n) = c$ for some constant $0 < c < \infty$, then $f(n) \in \Theta(g(n))$. ♠

Proof By the definition of limit, choose $\varepsilon = c/2$, there exists N such that

$$\frac{c}{2} = c - \varepsilon \leq \frac{f(n)}{g(n)} \leq c + \varepsilon = \frac{3c}{2}$$

for all $n \geq N$. It follows that $(c/2) \cdot g(n) \leq f(n) \leq (3c/2) \cdot g(n)$ for $n \geq N$. ■

Corollary 2.1

If $\lim_{n\to\infty} f(n)/g(n) = 0$, then $f(n) \in O(g(n))$ but not in $\Omega(g(n))$; if $f(n)/g(n)$ diverges when $n \to \infty$, then $f(n) \in \Omega(g(n))$ but not in $O(g(n))$. ♡

Property *Asymptotic bounds for some common functions:*

- *Polynomials: Let $f(n) = a_0 + a_1 n + \cdots + a_d n^d$ with $a_d > 0$, i.e., a polynomial of degree d, then $f(n) \in \Omega(n^d)$.*
- *Logarithms: Let $f(n) = \log_a n$ where $a > 1$, then $f(n) \in \Omega(\log_b n)$ for every $b > 1$.*
- *Logarithms and polynomials: Let $f(n) = \log_a n$ where $a > 1$, then $f(n) \in O(n^d)$ for every $d > 0$.*
- *Exponential and polynomials: Let $f(n) = n^d$ where $d > 0$, then $f(n) \in O(r^n)$ for every $r > 1$.*
- *Factorial: Let $f(n) = n!$, then $f(n) \in 2^{\Omega(n \log n)}$.*

Example 2.1

- Constant time (running time $O(1)$) example: condition branch, arithmetic/logic operation, declare variable, access element in an array, etc.
- Linear time (running time $O(n)$) example: iterate through an array, merge two sorted list in mergesort, etc.
- Logarithmic time (running time $O(\log n)$) example: binary search in a sorted array.
- Linearithmic time (running time $O(n \log n)$) example: mergesort.
- Quadratic time (running time $O(n^2)$) example: closest pair of points.
- Polynomial time (running time $O(n^k)$ where $k > 0$) example: fine k pairwise disjoint node in a graph.
- Exponential time (running time $O(2^{n^k})$ where $k > 0$) example: Euclidean TSP - find a tour of minimum length in a plane with n points.

Chapter 3 Graphs

3.1 Undirected Graph

3.1.1 Graph and Representations

Definition 3.1 (Undirected Graphs)

An undirected graphs is denoted by $G = (V, E)$, where V are nodes (vertices) and E are the edges between pairs of nodes, and we denote the size parameters as $n = |V|$ and $m = |E|$. It captures pairwise relationship between objects. ♣

Graph Representation: Adjacency Matrix The adjacency matrix is a $n \times n$ matrix with $A_{u,v} = 1$ is (u, v) is an edge.

- The space complexity is proportional to $\Theta(n^2)$.
- Checking if (u, v) is an edge takes $\Theta(1)$ time.
- Identifying all edges takes $\Theta(n^2)$ time.

Graph Representation: Adjacency List Node-index array of lists.

- The space complexity is proportional to $\Theta(n + m)$.
- Checking if (u, v) is an edge takes $O(\deg u)$ time, where $\deg u =$ number of neighbors of u.
- Identifying all edges takes $\Theta(m + n)$ time.

3.1.2 Path and Connectivity

Definition 3.2 (Path, Connectivity)

*A **path** in an undirected graph $G = (V, E)$ is a sequence of nodes $v_1, \cdots, v_k$ with the property that each consecutive edges is different. A path is **simple** if all nodes are distinct.*

*An undirected graph is **connected** if for every pair of nodes u, v, there is a path between u and v.* ♣

Definition 3.3 (Cycle)

*A **cycle** is a path $v_1, \cdots, v_k$ which $v_1 = v_k$ and $k > 1$. A cycle is simple if all nodes are distinct (except for v_1).* ♣

An undirected graph is a ***tree*** is it is connect and does not contain a cycle. A tree can be converted to a rooted tree if we choose a root r and orient all nodes away from r.

Property *Let G be an undirected graph on n nodes, any two of the following statements imply the third:*

- *G is connected*
- *G contains no cycles.*
- *G has $n-1$ edges.*

The ***connected components*** are all nodes reachable from s.

3.1.3 Graph Connectivity and Traversal

Two connectivity problems we want to solve:

1. s-t connectivity problem: given two nodes s and t, is there a path between s and t?

(2) s-t shortest path problem: given twp nodes s and t, that is the length of a shortest path between s and t?

Breadth First Search (BFS) The intuition is that we explore outward from s in all possible directions, adding nodes one "layer" at a time. Let

- $L_0 = \{s\}$ and $L_1 =$ all neighbors of L_0
- Recursively defined $L_{i+1} =$ all neighbors of nodes in L_i where do not belong to an earlier layer.

Proposition 3.1

For each i, L_i consists of all nodes at distance exactly i from s. There is a path from s to t iff t appears in some layer. ♠

Property *Let T be a BFS tree, if there is an edge (u, v), then the levels of x and y differ by at most* 1.

Proposition 3.2

The above implementation of BFS runs in $O(m+n)$ time if the graph is given by its adjacency list. ♠

Proof The loop runs at most n times since each node is traversed at most once. For each node u we visit, we take $O(1)$ to access and $\deg n$ operations to process since there are $\deg u$ incident edges. Since the total time processing edges is $\sum_{u \in V} \deg u = 2m$ and time processing nodes is $O(n)$, it follows that the total time is $O(m+n)$. ■

Example 3.1 Flood fill: given green pixel in an image, change color of entire blob of neighboring green pixels to blue. The nodes are pixels, edges are neighboring green pixels, and the blob represents the connected component of green pixels.

Depth-First Search

DFS explores a path at a time and backtrack whenever the path ends (or the node is already marked explored). DFS runs in $O(m+n)$.

Algorithm 2 DFS

```
1: Mark $u$ as "explored" and add $u$ to $R$
2: for each node $(u, v)$ incident to $u$ do
3:     if $v$ is not marked "explored" then
4:         Recursively invoke DFS($v$)
5:     end if
6: end for
```

3.1.4 Bipartiteness

Definition 3.4 (Bipartite)

An undirected graph $G = (V, E)$ is ***bipartite*** *if the nodes can be colored in two colors such that every edge incidents nodes with different colors.* ♣

Proposition 3.3

If a graph G is bipartite, it cannot contain an odd length cycle. ♠

Proposition 3.4

Let G be a connected graph, and let $L_0, \cdots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds:

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle, and hence G is not bipartite. ♠

Remark That is, if G is bipartite, we can orients the graph into layers of nodes for which there is no edges join nodes in the same layer or non-adjacent layers.

Proof (i) Suppose no nodes joins two nodes in same layer. By BFS property, each edge joins nodes in adjacent levels, so coloring white on odd layers and blue on even layers yields the desired coloring.

(ii) Suppose (x, y) is an edge with x, y in same level L_j. Then the cycle $x \to s \to y \to x$ has a odd length $j + j + 1$. Hence G is not bipartite. ■

3.2 Directed Graph

3.2.1 Connectivity in Directed Graphs

In a directed graph $G = (V, E)$, each edge $(u, v) \in E$ leaves node u and enters node v (ordered pair).

Definition 3.5 (Strong Connectivity)

*Nodes u and v are **mutually reachable** if there is both a path from u to v and also a path from v to u.*

*A graph is **strongly connected** if every pair of nodes is mutually reachable.* ♣

Proposition 3.5

Let s be any node. G is strongly connected if and only if every node is reachable from s and s is reachable from every node. ♠

Proof ($\Rightarrow$) This direction is trivial by definition. ($\Leftarrow$) For every pair of points u, v, concatenate the path $u \to s$ and $s \to v$ gives a path from u to v. The converse holds without loss of generality. ■

Theorem 3.1

The strong connectivity can be determined in $O(m + n)$ time. ♡

Proof Pick any node s, run BFS from s in G and run BFS from s in G^{reverse}, and return true iff all nodes reached in both BFS. The correctness follows from the previous lemma.

A ***strong component*** is a maximal subset of mutually reachable nodes. We can find all strong components in $O(m + n)$ time [Tarjan, 1972].

3.2.2 DAG and Topological Ordering

Definition 3.6 (DAG, Topological order)

*A **DAG** (**directed acyclic graph**) is a directed graph that contains no directed cycles. A **topological order** of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, \cdots, v_n$ so that for every edge (v_i, v_j) we have $i < j$.* ♣

Proposition 3.6

If G has a topological order, then G is a DAC. ♠

Proof For the sake of contradiction, suppose G has a topological order and a directed cycle C. Let v_i be the lowest indexed node and let (v_j, v_i) be an edge in C. Then by the definition topological order, $j < i$, contradicting the assumption. ■

Proposition 3.7

If G is a DAG, then G has a node with no entering edges. ♠

Proof Proof by contradiction. Suppose G is a DAG and every node has at least one entering edge. Fix a node v_1, choose v_2 where $(v_2, v_1) \in E$. Define recursively $v_3, \cdots, v_{n+1}$. $v_i = v_j$ for some $i < j$ by the Pigeonhole principle, then we obtain a cycle $v_j \to v_{j-1} \to \cdots \to v_i = v_j$, contradiction. ■

Proposition 3.8

If G is a DAG, then G has a topological ordering. ♠

Proof Suppose G is a DAG, choose v_1 be the node with no entering edges. $G - \{v_1\}$ is a DAG, since deleting v_1 cannot create cycles. Then we can recursively define $v_2, \cdots, v_n$ as above. The ordering $v_1, \cdots, v_n$ is the topological order of G, since v_i has no entering edges from $\{v_{i+1}, \cdots, v_n\}$. ■

Remark The topological order is not necessarily unique.

Theorem 3.2

Algorithm finds a topological order in $O(m + n)$ time. ♡

Chapter 4 Greedy Algorithm

4.1 Motivating Examples

4.1.1 Coining Changing

Consider the coining changing problem, where we have coin denomination of $\{1, 5, 10, 25, 100\}$ cents and we want to pay x cents using fewest coins. We can use Cashier's Algorithm: at each iteration, add coin of largest value that doesn't take us past the amount.

In general, suppose we have coin denomination $\{c_1, \cdots, c_n\}$ and amount x to pay, the Cashier's Algorithm is described as:

Cashiers' Algorithm

```
Sort $n$ coin demoniations so that $0 < c_1 < \cdots < c_n$.
$S \leftarrow \varnothing$
while $x > 0$ do
    $k \leftarrow$ largest coin denomination $c_k$ such that $c_k \leq x$.
    if no such k then
        return No solution.
    else
        $x \leftarrow x - c_k$
        $S \leftarrow S \cup \{k\}$.
    end if
end while
return $S$.
```

Note *Cashier's algorithm is optimal for U.S. coins* $\{1, 5, 10, 25, 100\}$.

Remark The Cashier's Algorithm is optional for U.S. coin denominations because of the special property of this denomination. However, this algorithm does not work in general.

4.1.2 Interval Scheduling

Consider the interval scheduling problem: we have job j starts at s_j and finishes at f_j, two jobs are compatible if they do not overlap. The goal is to find maximum subset of mutually compatible jobs.

```
Earliest-Finish-Time-First
1: Sort jobs by finish times and renumber so that $f_1 \leq \cdots \leq f_n$.
2: $S \leftarrow \varnothing$
3: for $j = 1$ to $n$ do
4:     if job $j$ is compatible with $S$ then
5:         $S \leftarrow S \cup \{j\}$.
6:     end if
7: end for
8: return $S$.
```

Note *The earliest-finish-time-first algorithm is optimal.*

Proof Proof by contradiction. Suppose greedy solution $i_1, \cdots, i_m$ is not the optimal solution, where the latter is denoted by $j_1, \cdots, j_r$. We can replace j_1 by i_1, since i_1 finish no later than j_1. Recursively, we can replace j_k by i_k in $\{j_1, \cdots, j_{k-1}, i_k\}$. This implies that $\{j_1, \cdots, j_r\}$ is also optimal, contradiction. ■

4.1.3 Interval Partitioning

Consider lecture j starts at s_j and finished at f_j, the goal is to find minimum number of classrooms to schedule all lectures.

```
Earliest-Start-Time-First
1: Sort lectures by starts times and renumber so that $s_1 \leq \cdots \leq s_n$.
2: $d \leftarrow 0$.
3: for $j = 1$ to $n$ do
4:     if lecture $j$ is compatible with some classroom then
5:         Schedule lecture $j$ in any such classroom $k$.
6:     else
7:         Allocate a new class room $d + 1$.
8:         Schedule lecture $j$ in classroom $d + 1$.
9:         $d \leftarrow d + 1$.
10:    end if
11: end for
12: return schedule.
```

Definition 4.1 (Depth)

*The **depth** of a set of open interval is the maximum number of intervals that contain any given point.* ♣

Property *Notice that the number of classrooms need to be greater than the depth. In addition, the above algorithm never schedules two compatible lecture in the same classroom.*

Note *The earliest-start-time-first algorithm is optimal.*

Proof Let d = number of classrooms allocated. Let classroom d to opened to schedule a lecture, j, that is incompatible with lectures in all other classrooms. Thus, the depth is at least d, since we have d lectures at $s_j + \varepsilon$.

Thus, all schedule uses at least d classrooms. ■

4.1.4 Minimize Maximal Lateness

Single resource processes on job at a time. Suppose job j requires t_j units of processing time and is due at time d_j; and if j starts at time s_j, it finishes at time $f_j = s_j + t_j$. We define the *lateness* to be $l_j = \max\{0, f_j - d_j\}$. The goal is to schedule all jobs to minimize *maximum lateness* $L = \max l_j$.

```
Earliest-Deadline-First
1: Sort lectures by deadlines and renumber so that $d_1 \leq \cdots \leq d_n$.
2: $t \leftarrow 0$.
3: for $j = 1$ to $n$ do
4:     Assign job $j$ to interval $[t, t + t_j]$.
5:     $s_j \leftarrow t$; $f_j \leftarrow t + t_j$
6:     $t \leftarrow t + t_j$
7: end for
8: return intervals $[s_1, f_1], \cdots, [s_n, f_n]$.
```

Define an ***inversion*** to be a pair of jobs i and j such that $d_i < d_j$ but j is scheduled before i.

Property

(1) There exists an optimal schedule with no idle time.

(2) The earliest-deadline-first schedule has no idle time.

(3) The earliest-deadline-first schedule is unique idle-free schedule with no inversions.

(4) If an idle-free schedule has an inversion, then it has an adjacent inversion (two inverted jobs scheduled consecutively).

Property *Exchanging two adjacent inverted jobs i and j reduced the number of inversion by 1 and does not increase the max lateness.*

Proof The first part of the statement is trivial. Let l, l' be the lateness before and after the swap, respectively. $l'_k = l_k$ for all $k \neq i, j$, and $l'_i \leq l_i$. Note that if j is late, $l'_j = f'_j - d_j = f_i - d_j \leq f_i - d_i \leq l_i$. ■

4.2 Dijkstra's Algorithm

Problem

- Single-pair Shortest Path Problem: Given a graph $G = (V, E)$, edge length $l_e \geq 0$, source $s \in V$, and destination $t \in V$, find a shortest directed path from s to t.
- Single-source Shortest Path Problem: Given a graph $G = (V, E)$, edge length $l_e \geq 0$, and source $s \in V$, find a shortest directed path from s to every node.

Dijkstra's Algorithm (Single-source) Maintain a set of explored nodes S for which algorithm has determined $d[u] =$ length of a shortest $s \to u$ path.

- Initialize $S \leftarrow \{s\}$, $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

$$\pi(v) = \min_{e=(u,v):u \in S} (d[u] + l_e),$$

  add v to S and set $d[v] \leftarrow \pi[v]$.
- To recover path, set pred$[v] \leftarrow e$ that achieves the minimum.

Proposition 4.1

Invariant: For each node $u \in S$, $d[u]$ is the length of a shortest $s \to u$ path. ♠

Proof We proceed by induction on $|S|$. $|S| = 1$ is trivial. Assume for $|S| \geq 1$, let v be the next node added to S. Suppose the Dijkstra path is $s \to u \to v$, and there is another path $p : s \to x \to v$. If $x \notin S$, $w_p > \pi(x) > \pi(v)$. If $x \in S$, then by the construction of v, $\pi(v) = \pi(u) + l_{u,v} \leq \pi(x) + l_{x,v} = l(p)$. Therefore, $\pi(v) \leq l(p)$. ■

Optimizations

(1) For each unexplored node $v \notin S$, we can explicitly maintain $\pi[v]$ instead of computing direction from definition, that is, update $\pi[v] \leftarrow \min\{\pi[v], \pi[u] + l_e\}$ if $e = (u, v)$.

(2) Use a min-oriented priority queue to choose an unexplored node that minimized $\pi[v]$.

Algorithm 3 Dijkstra's Algorithm

```
1:  $\pi[s] \leftarrow 0$; and for all $v \neq s$: $\pi[v] \leftarrow \infty$, pred$[v] \leftarrow$ null.
2:  Create an empty priority queue $pq$.
3:  for all $v \in V$: insert$(pq, v, \pi[v])$.
4:  while $pq$ is not empty do
5:      $u \leftarrow$ del-min$(pq)$.
6:      for all edge $e = (u, v) \in E$ leaving $u$ do
7:          if $\pi[v] > \pi[u] + l_e$ then
8:              decrease-key$(pq, v\pi[u] + l_e)$.
9:              $\pi[v] \leftarrow \pi[u] + l_e$; pred$[v] \leftarrow e$.
10:         end if
11:     end for
12: end while
```

Proposition 4.2

The time complexity for Dijkstra's algorithm is $O(|E| \log |V|)$. ♠

4.3 Minimum Spanning Trees

4.3.1 Minimum Spanning Trees

Definition 4.2 (Cut, Cutset)

*A **cut** is a partition of nodes into two nonempty subsets S and $V - S$. The cutset of a cut S is the set of edges with exactly one endpoint in S.* ♣

Proposition 4.3

A cycle and a cutset intersect in an even number of edges. ♠

Remark The key idea is that in a cycle, for every edge goes from S to $V - S$, there is another returning path.

Definition 4.3 (Spanning Tree, MST)

*Let $H = (V, T)$ be a subgraph of an undirected graph $G = (V, E)$. H is a **spanning tree** of G if H is both acyclic and connected.*

*Suppose each edge e corresponds a cost c_e, a **minimum spanning tree** (MST) (V, T) is a spanning tree of G such that the sum of the dges costs in T is minimized.* ♣

Property *Let $H = (V, T)$ be a subgraph of an undirected graph $G = (V, E)$, then the following are equivalent:*

- *H is a spanning tree of G, namely H is acyclic and connected.*
- *H is minimally connected: removal of any edge disconnects it, that is, connected and has $|V| - 1$ edges.*
- *H is maximally acyclic: addition of any edge creates a cycle, that is, acyclic and has $|V| - 1$ edges.*

Fundamental cycle For any non-tree edge $e \in E$, $T \cup \{e\}$ contains an unique cycle, denoted C. For any edge $f \in C$, $(V, T \cup \{e\} - \{f\})$ is a spanning tree.

Fundamental cutset For any tree edge $f \in T$, $(V, T = \{f\})$ has two connected components. Let D denote corresponding cutset, for any edge $e \in D$, $(V, T - \{f\} \cup \{e\})$ is a spanning tree.

Note *If $c_e < c_f$, then (V, T) is not an MST. This implies that we may use the exchange argument to construct the algorithms.*

The Greedy Algorithm

(i) **Red Rule**: Let C be a cycle with no red edges, color the uncolored edge with max cost to red.

(ii) **Blue Rule**: Let D be a cutset with no blue edges, color the uncolored edge with min cost to blue.

(iii) Apply red and blue rule until all edges are colored (we may stop once $n - 1$ edges colored blue), the blue edges form an MST.

Proposition 4.4

Color Invariant: There exists an MST (V, T^*) containing every blue edge and no red edge,

Proof We proceed by induction on number of iterations. Base case is trivial. Induction step (blue rule): choose a cut set D, let f denotes the edge colored blue. Proof by contradiction. Assume $f \notin T$, consider a fundamental cycle by adding f to T. Let $e \in T$ be another edge in D, then e is uncolored and $c_f \leq c_e$ (by blue rule), so $T \cup \{f\} - \{e\}$ is a spanning tree less than T. The proof for red rule is similar.

Proposition 4.5

The greedy algorithm terminates. Blue edges form an MST.

Proof Suppose an edge e is uncolored. If both endpoints are in same blue tree, we can apply red rule to cycle formed by e and the blue forest. Otherwise if endpoints of e are in different blue trees, we apply blue rule to cutset induced by one of the trees.

4.3.2 Prim's Algorithm

Note *Initialize $S = \{s\}$ for any node s, $T = \varnothing$. Repeat $n-1$ times:*

- *Add to T a min cost edge with exactly one endpoint in S.*
- *Add the other endpoints to S.*

Algorithm 4 Prim's Algorithm

```
1:  $S \leftarrow \varnothing, T \leftarrow \varnothing$
2:  $s \leftarrow$ an arbitrary node in $V$.
3:  $\pi[s] \leftarrow 0$; for all $v \neq s$ do $\pi[v] \leftarrow \infty$, pred$[v] \leftarrow$ null.
4:  Create an empty priority $pq$.
5:  for all $v \in V$ do insert$(pq, v, \pi[v])$.
6:  while $pq$ is nonempty do
7:      $u \leftarrow$ del-min$(pq)$.
8:      $S \leftarrow S \cup \{u\}, T \leftarrow T \cup \{$pred$[u]\}$.
9:      for all edge $e = (u, v) \in E$ with $v \notin S$ do:
10:         if $c_e < \pi[v]$ then
11:             decrease-key$(pq, v, c_e)$.
12:             $\pi[v] \leftarrow \pi[v] + c_e$; pred$[v] \leftarrow e$.
13:         end if
14:     end for
15: end while
```

Proposition 4.6 (Prim's Algorithm)

Prim's algorithm computes an MST.

Proof Special case of greedy algorithm (blue rule repeatedly applied to S).

Proposition 4.7

Prim's algorithm can be implemented to run in $O(m \log n)$ time.

4.3.3 Kruskal's Algorithm

Note *Consider edges in ascending order of cost, add to tree unless it would create a cycle.*

Algorithm 5 Kruskal's Algorithm

```
1: Sort $m$ edges by cost and renumber so $c_{e_1} \leq c_{e_2} \leq \cdots$.
2: $T \leftarrow \varnothing$.
3: for all $v \neq V$ do make-set($v$).
4: for $i = 1$ to $m$ do
5:     $(u, v) \leftarrow e_i$.
6:     if find-set($u$) $\neq$ find-set($v$) then
7:         $T \leftarrow T \cup \{e_i\}$.
8:         union($u, v$)                    ▷ Make $u$ and $v$ in same component.
9:     end if
10: end for
11: return $T$.
```

Proposition 4.8

Kruskal's algorithm computes an MST.

Proposition 4.9

Kruskal's algorithm can be implemented to run in $O(m \log m)$ time.

We sort edges by cost, and use *union-find* data structure to dynamically maintain connected components.

4.3.4 Reverse-delete Algorithm

Note *Start with all edges in T and consider them in descending order of cost. Delete edge from T unless it would disconnect T.*

Proposition 4.10

The reverse-delete-algorithm computes an MST.

Proof Special case of greedy algorithm, where we apply red rule to a cycle containing e if deleting e does not disconnect T, and apply blue rule to the cutset induced by either component (e will become blue since it is the only remaining edge).

4.3.5 Boruvka's Algorithm

Note *Start by considering every node itself as a tree, repeat until only one tree:*

- *Apply blue rule to cutset corresponding to each blue tree.*
- *Color all selected edges blue.*

Proposition 4.11

Boruvka's algorithm computes the MST if edge costs are distinct.

Proof Special case of greedy algorithm (repeatedly apply blue rule).

Chapter 5 Divide and Conquer

Divide-and-conquer approaches

- Divide up problem into several subproblems
- Solve (conquer) each subproblem recursively
- Combine solutions to subproblems into overall solution.

5.1 Motivating Examples

5.1.1 Mergesort

Divide-and-conquer approach: Mergesort recursively sort left and right half and merge two halves to make sorted whole.

Algorithm 6 Mergesort, Merge-Sort(L)

```
1: if list $L$ has one element then
2:     return $L$.
3: end if
4: Divide the list into two halves $A$ and $B$.
5: $A \leftarrow$ Merge-Sort($A$).                ▷ $T(n/2)$
6: $B \leftarrow$ Merge-Sort($B$).                ▷ $T(n/2)$
7: $L \leftarrow$ Merge($A, B$).                  ▷ $\Theta(b)$
8: return $L$.
```

$T(n) =$ max number of compares to mergesort a list of length n. The recurrence is

$$T(n) \le \begin{cases} 0 & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n & \text{if } n > 1 \end{cases}.$$

where n represents the comparison between sets in the partition. Assume n is a power of 2 and $n > 1$, the $T(n) = 2(T/2) + n$. Then it is not hard to prove $T(n) = n \log n$ by induction.

Proposition 5.1

$T(n) \le n\lceil \log n \rceil$.

Proof We proceed by strong induction on n. The base case $n = 1$ is trivial. For $n > 1$, define $n_1 = \lfloor n/2 \rfloor$ and $n_2 = \lceil n/2 \rceil$ and note that $n = n_1 + n_2$. Then

$$\begin{aligned} T(n) &\le T(n_1) + T(n_2) + n \le n_1 \lceil \log n_1 \rceil + n_2 \lceil \log n_2 \rceil + n \\ &\le n\lceil \log n_2 \rceil + n \le n(\lceil \log n \rceil - 1) + n \\ &= n\lceil \log n \rceil. \end{aligned}$$

Sorting Lower Bound

Proposition 5.2

An deterministic compare-based sorting algorithm must make $\Omega(n \log n)$ compares in the worst-case. ♠

Proof There are $n!$ different orderings of an array consists of n distinct values, and note that a binary tree of height h has no more than 2^h leaves. By the Stirling approximation,

$$2^h \geq n! \quad \Longrightarrow \quad h \geq \log(n!) \geq n \log n - n/\ln 2,$$

so there are $\Omega(n \log n)$ comparisons in the worst-case. ■

5.1.2 Counting Inversions

Suppose there is a two rank on n objects. Rank A: $1, \cdots, n$, and rank B: $a_1, \cdots, a_n$. Objects i and j are inverted if $i < j$ but $a_i > a_j$.

Divide-and-conquer:

- Divide: separate list into halves A and B
- Conquer: recursively count inversions in each list.
- Combine: count inversions (a, b) with $a \in A$ and $b \in B$.

However, combine two subproblems may be challenging. The $O(n)$ approach is, assuming A and B are sorted, to scan and compare A and B from left to right, compare elements to update inversions, and update points.

Counting Inversion, Sort-and-Count(L)

```
1: if list $L$ has one element then
2:     return $(0, L)$.
3: end if
4: Divide the list into two halves $A$ and $B$.
5: $(r_A, A) \leftarrow$ Sort-and-Count($A$).                    ▷ $T(n/2)$
6: $(r_B, B) \leftarrow$ Sort-and-Count($B$).                    ▷ $T(n/2)$
7: $(r_{AB}, L) \leftarrow$ Merge-and-Count($A, B$).   ▷ Merge the sorted sets and count the inversions, $\Theta(n)$
8: return $(r_A, r_B, r_{AB}, L)$.
```

Proposition 5.3

The sort-and-count algorithm counts the number of inversions in a permutation of size n in $O(n \log n)$ time. ♠

5.1.3 Randomized Quicksort

Three-way Partitioning

- Goal: Given an array A and fixed a pivot p, partition the array so that elements smaller than p are in the left subarray L, elements equal to p are in the middle subarray M, and element larger than p are in right subarray R.

- Approach: Let the first item p of A be the pivot; place points l, r on the front and end of the array, respectively. Scan i from front to end,
    - If $A[i] < p$: swap $A[l]$ and $A[i]$, increment both l and i.
    - If $A[i] > p$: swap $A[r]$ and $A[i]$, decrement r.
    - If $A[i] = p$: increment i.

Algorithm 7 Randomized-Quicksort(A)

```
1: if array $A$ has zero or one element then
2:     return
3: end if
4: Pick pivot $p \in A$ uniformly at random.
5: $(L, M, R) \leftarrow$ Partition-3-way(A, p).        ▷ $\Theta(n)$
6: Randomized-Quicksort($L$)                         ▷ $T(i)$
7: Randomized-Quicksort($R$)                         ▷ $T(n-i-1)$
```

Proposition 5.4

The expected number of compares to quicksort an array of n distinct elements is $O(n \log n)$.

Proof Suppose $a_1 < \cdots < a_n$. The probability of a_i and a_j are compared is $2/(j-i+1)$ [a_i and a_j is compared iff a_i or a_j is selected and is not compared iff $a_k : i < k < j$ is selected as pivot]. Thus, the expected number of compares is

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = 2 \sum_{i=1}^{n} \sum_{j=2}^{n-i+1} \frac{1}{j} \leq 2n \sum_{j=1}^{n} \frac{1}{j} \leq 2n(\ln n + 1).$$

where the last inequality holds from the harmonic sum. ■

5.1.4 Closed Pair of Points

Problem Given n points in the plane, find a pair of points with the smallest Euclidean distance between them. We may use non-degeneracy assumption: no two points have the same x-coordinate.

Divide and Conquer

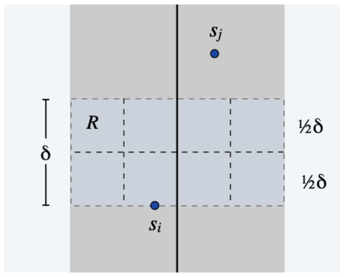
- Divide: draw vertical line L so that $n/2$ points on each side.
- Conquer: find closest pair in each side recursively
- Combine: find closed pair with one point in each side.
- Return: the best of three.

The "combine" step takes $\Theta(n^2)$ using brute force. Indeed, assuming the shortest distance is δ in the conquer step, it suffices to consider points within δ of line L, called the 2δ-strip.

Let s_i be the point in the 2δ-strip, ordering determined by ascending y-coordinate.

Property *If $|j-i|>7$, then the distance between s_i and s_j is at least δ.*

Proof Consider 2δ by δ rectangle R in strip whose min y-coordinate is y-coordinate of s_i. Subdivide R into 8 squares, and there is at most 1 point per each square. Then $|j-i|>7$ implies $s_j \notin R$, so there distance is at least δ.



```
Closest-Pair($p_1, \cdots, p_n$)
1: ▷ Divide and Conquer
   Compute vertical line $L$ such that half the points are on each side of the line.   ▷ $O(n)$
2: $\delta_1 \leftarrow$ Closest-Pair( points in the left half ).                       ▷ $T(n/2)$
3: $\delta_2 \leftarrow$ Closest-Pair( points in the right half ).                      ▷ $T(n/2)$
4: ▷ Combine
   $\delta \leftarrow \min\{\delta_1, \delta_2\}$.
5: Delete all points further than $\delta$ from $L$.                                    ▷ $O(n)$
6: Sort remaining points by $y$-coordinate.                                             ▷ $O(n \log n)$
7: Scan points in $y$-order and compare distance between each point and next 7 neighbors.
   If any of these distances is less than $\delta$, update $\delta$.                    ▷ $O(n)$
8: return $\delta$.
```

5.2 Master's Theorem

Theorem 5.1 (Master's Theorem)

Let $a \geq 1$, $b \geq 2$, and $c \geq 0$ and suppose that $T(n)$ is a function on the nonnegative intgers that satisfies the recurrence

$$T(n) = aT(n/b) + \Theta(n^c)$$

with $T(0) = 0$ and $T(1) = \Theta(1)$, where n/b means either the floor or ceiling of n/b.

(1) Case 1: If $c > \log_b a$ (i.e., $b^c > a$), then $T(n) = \Theta(n^c)$.

(2) Case 2: If $c = \log_b a$ (i.e., $b^c = a$), then $T(n) = \Theta(n^c \log n)$.

(3) Case 3: If $c < \log_b a$ (i.e., $b^c < a$), then $T(n) = \Theta(n^{\log_b a})$. ♡

Example 5.1 The integral multiplication runs in $\Theta(n^2)$ using grade-school algorithm. However, the Karatsuba trick divides x, y into low and high order bits $x = 2^m a + b$ and $y = 2^m c + d$, then $xy = 2^{2m}(ac) + 2^m(ad + bc) + bd$ (brute force using this identity still requires $\Theta(n^2)$). Indeed, we can compute the middle term by $ad + bc = ac + bd - (a - b)(c - d)$, and ac, bd is free in terms of complexity. Thus, $T(n) = 3T(n/2) + \Theta(n)$, so by Master's Theorem $T(n) = \Theta(n^{\log_2 3}) \approx O(n^{1.585})$. Hence the grade-school multiplication is not optimal.

Chapter 6 Dynamic Programming

Dynamic Programming Break up a problem into series of *overlapping* subproblems (whereas in divide-and-conquer, the subproblems are independent); combine solutions to smaller subproblems to form solution to large subproblem. Dynamic programming is planning over time, we catch intermediate results in a table for later reuse.

6.1 Motivating Examples

6.1.1 Weighted Interval Scheduling

Recall in unweighted interval scheduling problem (i.e., all weights are 1), we can use greedy approach: consider jobs in ascending order of finish time. However, the greedy algorithm is no longer applicable if jobs are weighted.

Dynamic Programming Let jobs be sorted in ascending order of finish time $f_1 \leq \cdots \leq f_n$. Denote by $p(j)$ the largest index $i < j$ such that job i is compatible with j. We want to optimize $\text{OPT}(j)$ = optimal weight of subproblem consisting only jobs $1, \cdots, j$.

- Case 1: $\text{OPT}(j)$ does not select job j. Then $\text{OPT}(j) = \text{OPT}(j-1)$, i.e., we employs the optimal solution of its subproblem.
- Case 2: $\text{OPT}(j)$ selects job j. Jobs compatible with j are $1, \cdots, p(j)$, so $\text{OPT}(j) = w_j + \text{OPT}(p(j))$.

We therefore obtain the Bellman equation (optimizing equation)

$$\text{OPT}(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{\text{OPT}(j-1), w_j + \text{OPT}(p(j))\} & \text{if } j > 0 \end{cases}.$$

Remark The brute-force approach using Bellman equation causes repeated computation, resulting in exponential-time algorithm. In dynamic programming, we use the idea of *memorization* to cache the result of subproblem j in $M[j]$.

Weighted Interval Scheduling (Bottom-up)

```
1: Sort jobs by finish time and renumber so that $f_1 \leq \cdots \leq f_n$.
2: Compute $p[1], \cdots, p[n]$ via binary search.
3: $M[0] = 0$                                                    ▷ Global array
4: for $j = 1, \cdots, n$ do
5:     $M[j] \leftarrow \max\{\text{OPT}(j-1), w_j + \text{OPT}(p(j))\}$.
6: end for
7: return $M[n]$.
```

Remark The top-down approach is also applicable: we can call $\text{OPT}(n)$ directly and use recursion to incur the subproblems when calculating $M[n]$.

Proposition 6.1

The algorithm above takes $O(n \log n)$ time.

Note *To obtain the optimal solution (scheduling of jobs), we can compare $M[j]$ to $M[j-1]$ to determine whether job j is selected.*

6.1.2 Segmented Least Squares

Problem Given n points in the plane, we want to find a reasonable choice for $f(x)$ (sequence of line segments) to balance accuracy and parsimony (number of lines). In other words, the goal is to minimize $E + cL$ for some constant $c > 0$, where E is the sum of squared errors, L is the number of lines.

Dynamic Programming: Multiway Choice Define $\text{OPT}(j)$ = minimum cost for points, and $e_{ij} = SSE$ for points $p_i, p_{i+1}, \cdots, p_j$. If the last segment uses points $p_i, \cdots, p_j$, the cost is given by $e_{ij} + c + \text{OPT}(i-1)$. The Bellman equation is

$$\text{OPT}(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{1 \le i \le j} \{e_{ij} + c + \text{OPT}(i-1)\} & \text{if } j > 0 \end{cases}.$$

Proposition 6.2 (Bellman 1916)

DP algorithm solves the segmented least squares problem in $O(n^3)$ time and $O(n^2)$ space.

The time complexity can be improved to $O(n^2)$ by precompute cumulative sums $\sum_{k=1}^{i} x_k$, $\sum_{k=1}^{i} y_k$, $\sum_{k=1}^{i} x_k^2$, and $\sum_{k=1}^{i} x_k y_k$. Using cumulative sums, we can compute e_{ij} in $O(1)$ times.

6.2 Knapsack Problem

Problem Suppose the knapsack has weight limit of W, and there are n items: item i provides value $v_i > 0$ and weights $w_i > 0$. We want to pack knapsack so as to maximize total value of items taken.

Dynamic Programming Define OPT(i, w) = optimal of knapsack problem with items $1, \cdots, i$ subject to weight limit w. The goal is to optimize OPT(n, W).

- Case 1: OPT(i, w) does not select item i, then OPT$(i, w) =$ OPT$(i-1, w)$.
- Case 2: OPT(i, w) selects item i, then OPT$(i, w) =$ OPT$(i-1, w-w_i)$.

Thus the Bellman equation is

$$\text{OPT}(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ \text{OPT}(i-1,w) & \text{if } w_i > w \\ \max\{\text{OPT}(i-1,w), v_i + \text{OPT}(i-1,w-w_i)\} & \text{otherwise} \end{cases}.$$

Algorithm 8 Knapsack Problem (Bottom-up)

```
for each $w = 0, \cdots, W$ do $M[0,w] \leftarrow 0$.
for $i = 1, \cdots, n$ do
    for $w = 0, \cdots, W$ do
        if $w_i > w$ then
            $M[i,w] \leftarrow M[i-1,w]$.
        else
            $M[i,w] \leftarrow \max\{M[i-1,w], v_i + M[i-1,w-w_j]\}$.
        end if
    end for
end for
return $M[n,W]$.
```

Proposition 6.3

The DP algorithm solves the knapsack problem with n items and maximum weight W in $\Theta(nW)$ time and $\Theta(nW)$ space.

6.3 Dynamic Programming in Graphs

6.3.1 Bellman-Ford-Moore Algorithm

Problem Given a digraph $G = (V, E)$ with arbitrary length (not necessarily nonnegative) c_{vw} associated with each edge, find the shortest path $v \rightsquigarrow t$.

Issue With Dijkstra Consider the graph where the edge weights c_{vw} are arbitrary (can be negative), Dijkstra may not produce shortest paths.

Definition 6.1 (Negative Cycles)

*A **negative cycle** is a directed cycle for which the sum of its edge lengths is negative.* ♣

Proposition 6.4

(a) If some $v \rightsquigarrow t$ path contains a negative cycle, then there does not exist a shortest $v \rightsquigarrow t$ path.

(b) If G has no negative cycles, then there exists a shortest $v \rightsquigarrow t$ path that is simple (and thus has at most $n - 1$ edges). ♠

Dynamic Programming Solution Define $\text{OPT}(i, v)$ to be the minimum cost of $v \rightsquigarrow t$ path using at most i edges.

- Case 1: At most $i - 1$ edges are used, then $\text{OPT}(i - 1, v)$.
- Case 2: If i edges are used, and (v, w) edge is in the path, $\text{OPT}(i, v) = c_{vw} + \text{OPT}(i - 1, w)$.

Then the Bellman equation is given by

$$\text{OPT}(i, v) = \min \left\{ \text{OPT}(i - 1, v), \min_{(v,w)\in E} \{c_{vw} + \text{OPT}(i - 1, w)\} \right\}.$$

Algorithm 9 Shortest-Path(G, s, t)

```
1: $n \leftarrow$ number of nodes in $G$.
2: Define an array $M[\{0, \cdots, n-1\}, V]$ by the equalities $M[0, t] = 0$ and $M[0, v] = \infty$ for all $v \neq t$.
3: for $i = 1, \cdots, n - 1$ do
4:     for $v \in V$ do
5:         Compute $M[i, v]$ using the recurrence.
6:     end for
7: end forreturn $M[n - 1, s]$.
```

Proposition 6.5

Given a directed graph $G = (V, E)$ with no negative cycles, the DP algorithm computes the length of a shortest $v \rightsquigarrow t$ path for every node v in $\Theta(mn)$ time and $\Theta(n^2)$ space. ♠

6.3.2 Detecting Negative Cycles

Problem Given a digraph $G = (V, E)$, with edge length c_{vw}, find a negative cycle if one exists.

Lemma 6.1

There exists a node $v \in V$ such that OPT$(n, v) <$ OPT$(n-1, v)$ if and only if (any) shortest $v \rightsquigarrow t$ path of length $\leq n$ contains a cycle W. Moreover W is a negative cycle. ♡

Proposition 6.6

Can find whether a graph contains a negative cycle in $\Theta(mn)$ and $\Theta(n^2)$ space. ♠

Chapter 7 Network Flow

7.1 Max-flow and Min-cut Problem

7.1.1 Problem Introduction

Max-flow Problems Find a flow of maximum value.

Definition 7.1 (Flow Network)

*A **flow network** is a tuple $G = (V, E, s, t, c)$. It contains a digraph (V, E) with source $s \in V$ (assume all nodes are reachable from s) and sink $t \in V$. Capacity $c(e) \geq 0$ (assume integers) for each $e \in E$.*

Definition 7.2 (st-flow)

*An **st-flow** (**flow**) is a function that satisfies:*

- *(Capacity) For each $e \in E$, $0 \leq f(e) \leq c(e)$.*
- *(Flow conservation) For each $v \in V \setminus \{s, t\}$: $\sum_{e \text{ in to} v} f(e) = \sum_{e \text{ out of } v} f(e)$.*

*The **value** of a flow f is: $val(f) = \sum_{e \text{ out of} s} f(e) - \sum_{e \text{ in to } s} f(e)$.*

Minimum-cut Problem Find a cut of minimum capacity.

Definition 7.3 (st-cut)

*An **st-cut** (**cut**) is a partition (A, B) of the nodes with $s \in A$ and $t \in B$. The **capacity** of an st-cut is the sum of capacities of the edges from A to B, namely $cap(A, B) = \sum_{e \text{ out of } A} c(e)$.*

7.1.2 Ford–Fulkerson algorithm

Consider the greedy algorithm that finds paths and augment flow repeatedly. The issue of this algorithm is that it cannot decrease the flow to do backtracking on bad distribution.

Residual Network Suppose edge $e_0 = (u, v) \in E$ with flow $f(e_0)$ and capacity $c(e_0)$. We create an edge $e \in E_f$ where $c(e) = c(e_0) - f(e_0)$ and a reverse edge $e^{-1} = (v, u)$ with capacity $c(e^{-1}) = f(e_0)$ to undo the flow sent. For example, the edge $e_0 = 6/17$ is converted to e and e^{-1} as shown below

$$\begin{tikzcd} u \arrow[r, "e_0=6/17"] & v \end{tikzcd} \qquad \begin{tikzcd} u \arrow[r, bend left, "e=0/11"] & v \arrow[l, bend left, "e^{-1}=0/6"] \end{tikzcd}$$

Augmenting Path An ***augmenting path*** is a simple $s \rightsquigarrow t$ path in the residual network G_f, its ***bottleneck capacity*** is the minimum residual capacity of any edge in P.

Property *Let f be a flow and P be an augmenting path in G_f. The after calling* $\textsc{Augment}(f, c, P)$*, which augment the flow in residual network, the resulting f' is a flow and val(f') = val(f) + bottleneck(G_f, P).*

Ford-Fulkerson

- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an $s \rightsquigarrow t$ path in the residual network G_f.
- Augment flow along path P.
- Repeat until get stuck.

Algorithm 10 Ford-Fulkerson

```
function Ford-Fulkerson(G)
    for each edge $e \in E$ do $f(e) \leftarrow 0$.
    $G_f \leftarrow$ residual network of $G$ with respect to flow $f$.
    while there exists an $s \rightsquigarrow t$ path $P$ in $G_f$ do
        $f \leftarrow$ Augment$(f, c, P)$.
        Update $G_f$.
    end while
    return $G_f$.
end function

function Augment$(f, c, P)$
    $\delta \leftarrow$ bottleneck capacity of augmenting path $P$.
    for each $e \in P$ do
        if $e \in E$ them $f(e) \leftarrow f(e) + \delta$.
        else $f(e^{-1}) \leftarrow f(e^{-1}) - \delta$        ▷ if $e = d^{-1}$ for some $d \in E$, let $f(d) \leftarrow f(d) - \delta$
    end for
    return $f$.
end function
```

7.1.3 Max-flow and Min-cut Theorem

Proposition 7.1 (Flow Value Lemma)

Let f be any flow and let (A, B) be any cut. Then the value of the flow f equals the new flow across the cut (A, B), i.e., val$(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e)$. ♠

Proposition 7.2 (Weak Duality)

Let f be any flow and (A, B) be any cut. Then val$(f) \leq$ cap(A, B). ♠

Theorem 7.1 (Max-flow and Min-cut Theorem, Strong Duality)

The value of a max flow is equal to the capacity of a min cut. ♡

Proof It suffices to prove the following three conditions are equivalent:

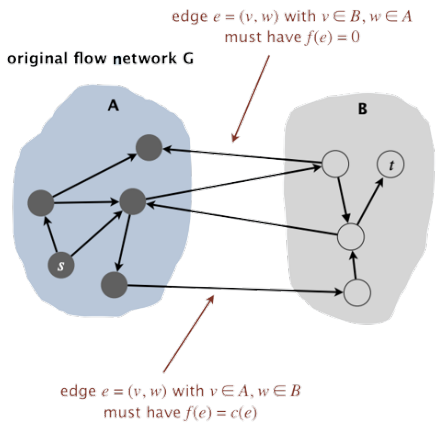
(a) There exists a cut (A, B) such that $\text{cap}(A, B) = \text{val}(f)$.

(b) f is a max flow.

(c) There is no augmenting path with respect to f.

$(a) \Rightarrow (b)$ follows immediately from weak duality. $(b) \Rightarrow (c)$ follows from contrapositive, having augmenting path means we can improve f by sending flow along this path.

$(c) \Rightarrow (a)$: Let f be a flow with no augmenting paths and let $A =$ set of nodes reachable from s in residual network G_f. It is obvious that $s \in A$ and $t \notin A$. Then $\text{val}(f) = \sum_{e \text{ out}} f(e) - \sum_{e \text{ in}} f(e) = c(e) - 0 = \text{cap}(A, B)$. ■


edge $e = (v, w)$ with $v \in B, w \in A$
must have $f(e) = 0$
original flow network G
A
B
s
t
edge $e = (v, w)$ with $v \in A, w \in B$
must have $f(e) = c(e)$

Given any max flow f, we can compute a min cut (A, B) in $O(m)$ times. To obtain the min cut, we can let $A =$ set of nodes reachable from s in residual network G_f, and A is the min cut according to the proof of above theorem.

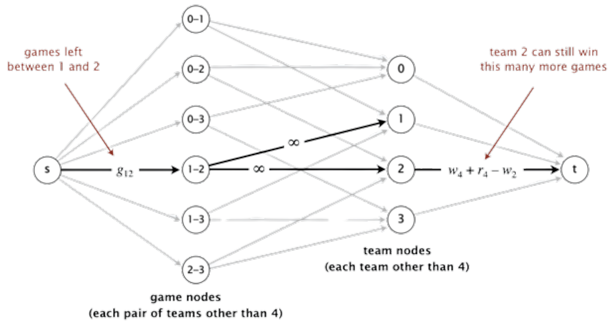
7.1.4 Application - Baseball Elimination Problem

Problem Given the set of teams S with a distinguished team $z \in S$. For each team x, given that x has won w_x games already, and team x and y play each other r_{xy} additional times for all $y \neq x$.

The problem is that: given current standings, is there any outcome of the remaining games in which team z finishes with the most (or tied for the most) wins?

7.1.4.0.1 Network Flow Solution We can create a network so that the first layer is a singleton s, the second layer represents matches between teams other than z, the third layers represents all teams other than z, and the last layer is a singleton t.

The edges from layer 1 to 2 have capacities equal to the number of corresponding games left, edges from layer 2 to 3 have infinity capacity, and edges from layer 3 to 4 have capacity of maximum number of wins of x given that team z wins.


games left between 1 and 2
team 2 can still win this many more games
s
0–1
0–2
0–3
1–2
1–3
2–3
g_{12}
∞
∞
0
1
2
3
t
$w_4 + r_4 - w_2$
team nodes (each team other than 4)
game nodes (each pair of teams other than 4)

Proposition 7.3

Team z is not eliminated if and only if max flow saturates all edges leaving s.

Proof Note that each remaining game between x and y added to number of wins for team x or team y. Capacity on (x, t) edges ensures no team wins too many games.

Note *Hoffman-Rivlin, 1967 Suppose $T \subset S$, denote by $w(T)$ the number of total wins of teams in T and by $g(T)$ the number of remaining games between teams in T,*

$$w(T) := \sum_{i \in T} w_i, \qquad g(T) := \sum_{(x,y) \subset T} g_{xy}.$$

Team z is eliminated if and only if there exists a subset T^* such that

$$w_z + g_z < \frac{w(T^*) + g(T^*)}{|T^*|}.$$

Remark That is, z is eliminated if and only if the number of maximum possible wins of z is less than the least average possible wins of some $T \subset S$.

7.1.5 Application - Bipartite Matching

Problem Given a bipartite graph $G = (L \cup R, E)$, find a max-cardinality matching.

Network Flow Solution Add a super source s and a super sink t, and let s be connected to every node in L and t be connected to every node in R, and assign capacity of 1 for every such edges. Let edges from L to R have infinite capacity. Denote the network G'.

Proposition 7.4

There is a bijection between matching of cardinality k in G and integral flow of value k in G'

Chapter 8 Intractability

8.1 Reductibility and Satisfiability

8.1.1 Reductibility

Definition 8.1 (Reduction)

*Problem X **polynomial-time reduces** to problem Y if arbitrart instances of probelm X can be solved using:*

- *polynomial number of standard computational steps, plus*
- *polynomial number of calls to oracle that solves problem Y.*

We use the notation $X \leq_p Y$. ♣

If $X \leq_p Y$ and Y can be solved in polynomial time, then X can be solved in polynomial time.

Note *If both $X \leq_p Y$ and $Y \leq_p X$, we use notation $X \equiv_p Y$; in this case, X can be solved in polynomial time iff Y can be.*

Note *The transitivity holds for $\leq_p$: if $X \leq_p Y$ and $Y \leq_p Z$, then $X \leq_p Z$.*

8.1.2 Packing and Covering Problems

Given a graph G and $k \in \mathbb{Z}$,

- Independent-Set: Is there a subset of k (or more) vertices such that no two are adjacent?
- Vertex-Cover: Is there a subset of k (or fewer) vertices such that each edge is incident to at least one vertex in the subset?

Given a set U of elements, a collection S of subsets of U, and an integer k

- Set-Cover: Are there $\leq k$ of these subsets whose union is equal to U?

Proposition 8.1

Independent-Set $\equiv_p$ Vertex-Cover ♠

Proof Sketch: S is an independent set of size k iff $V - S$ is a vertex cover of size $n - k$.

Proposition 8.2

Vertex-Cover $\leq_p$ Set-Cover. ♠

Proof Sketch: Let $S_v = \{e \in E \mid e \text{ incident to } v\}$ for all v. Suppose $\{S_{v_i}\}$ forms a set cover, then $\{v_i\}$ is a vertex cover, and vice versa. Hence G contains a vertex cover of size k iff U contains a set cover of size k. ■

8.1.3 Satisfiability

A literal is a boolean variable or its negation, namely x_i or $\bar{x}_i$, a clause is a disjunction ($\vee$) of literals, namely $C_j = x_1 \vee w_2 \vee \cdots$.

Definition 8.2 (Satisfiability)

The conjuctive normal form (CNF) is a propositional formula Φ that is a conjunction ($\wedge$) of clauses. The ***satisfiability (SAT)*** *problem is to determine whether there is a truth assignment that satisfies the given CNF formula Φ.* ***3-SAT*** *problem is the SAT problem where each clause contains exactly three literals (each literal corresponds to a different variable).* ♣

The scientific hypothesis that that there does not exists a poly-time algorithm for 3-SAT. This hypothesis is equivalent to $P \neq NP$ conjecture.

Proposition 8.3

3-SAT $\leq_p$ INDEPENDENT-SET. ♠

Proof Sketch: Given an instance Φ of 3-SAT, we can construct a graph G by including all literals as nodes, connect 3 literals in a clause, and connect literal to each of its negation. Then Φ is satisfiable iff G contains an independent set of size $k = |\Phi|$ (by setting literals in the independent set to true and everything else to false). ■

8.1.4 Sequencing Problem

HAMILTON-CYCLE: Given an undirected graph $G = (V, E)$, does there exist a cycle Γ that visits every node exactly once?

DIRECTED-HAMILTON-CYCLE: Given an directed graph $G = (V, E)$, does there exist a cycle Γ that visits every node exactly once?

Proposition 8.4

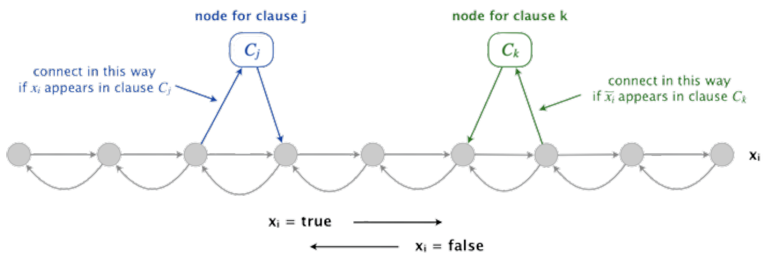
DIRECTED-HAMILTON-CYCLE $\leq_p$ HAMILTON-CYCLE. ♠

Proof Sketch: Given a directed graph G, we can construct a graph G' with $3n$ nodes: For each $v \in V$, construct v_{in}, v, v_{out} in G', and connect (v_{in}, v) and (v, v^{out}). For each $(v, v') \in E$, connected v_{out} and v'_{in}. Then G has a directed Hamilton cycle iff G' has a Hamilton cycle, since each (v_{out}, v'_{in}) in the Hamilton cycle of G' represents (v, v') in the directed Hamilton cycle of G. ■

Proposition 8.5

3-SAT $\leq$ DIRECTED-HAMILTON-CYCLE. ♠

Proof Sketch: Given 3-SAT instance Φ with n variable and k clauses. For each x_i, construct a double linked list of nodes in row i, let the traversal of path i from left to right be equivalent to setting x_i to true. For each clause C_j, add a node C_j and two edges, where the edges are connected left to right if x_i appears in C_j and connected right to left if $\bar{x}_i$ appears.


node for clause j
node for clause k
C_j
C_k
connect in this way if x_i appears in clause C_j
connect in this way if $\bar{x}_i$ appears in clause C_k
x_i
x_i = true
x_i = false

Then we claim Φ is satisfiable iff G has a Hamilton cycle. If Φ has a satisfying assignment x^*. Let row i be traversed from left to right iff $x_i^* =$ true; then for each clause, there will be at least one row i which are going in the correct direction to splice clause node C_j into cycle. Conversely, suppose Γ is a Hamilton cycle. By replacing the edge connecting to C_j by an parallel edge in row i, we obtain Hamilton cycle of $G - \{C_1, \cdots, C_k\}$. Set x_i^* as above, then each clause is satisfied. ■

8.1.5 Graph Coloring and Numerical Problems

3-Color: Given an undirected graph G, can nodes be colored in three colors so that no adjacent nodes have the same color.

Proposition 8.6

3-Sat $\leq_p$ 3-Color.

Subset-Sum: Given n natural numbers $w_1, \cdots, w_n$ and an integer W, is there a subset that adds up to exactly W.

Proposition 8.7

3-Sat $\leq_p$ Subset-Sum.

8.1.6 Summary

In conclusion, 3-Sat problem can be reduced to the above problems:

- 3-Sat $\leq_p$ Independent-Set $\equiv_p$ Vertex-Cover $\leq_p$ Set-Cover.
- 3-Sat $\leq_p$ Dir-Ham-Cycle $\leq_p$ Ham-Cycle.
- 3-Sat $\leq_p$ 3-Color.
- 3-Sat $\leq_p$ Subset-Sum.

8.2 P versus NP

8.2.1 P vs. NP

Definition 8.3 (P, NP)

Algorithm A runs in polynomial time if for every string s, $A(s)$ terminates in $\leq p(|s|)$ steps for some polynomial function $p(\cdot)$. ***P*** *$=$ set of decision problems for which there exists a poly-time algorithm.*

Algorithm $C(s,t)$ is a ***certifier*** *for problem X is for every string s: $s \in X$ iff there exists a string t such that $C(s,t) = true$.* ***NP*** *$=$ set of decision problems for which there exists a poly-time certifier.*

EXP *$=$ decision problems for which there exists an exponential-time algorithm.* ♣

Example 8.1 Examples of problems in P are: Prime (decide if x is a prime), L-Solve (decide if there is a vector x such that $Ax = b$), etc.

Examples of problems in NP are: Sat, Hamilton-Path, Factor (decide if x has a nontrivial factor less than y).

Proposition 8.8

$P \subseteq NP \subseteq EXP$. ♠

Proof P $\subseteq$ NP follows immediately by setting the certificate $t = \varepsilon$, i.e., $C(s,t) = A(s)$. Consider $X \in$ NP, there exists a poly-time certifier $C(s,t)$. To solve instance s, run $C(s,t)$ on all strings t with $|t| \leq p(|s|)$, then the answer exists iff $C(s,t)$ is true for some certificates. ■

8.2.2 NP-Complete

Definition 8.4 (NP-Complete)

A problem $Y \in$ ***NP-C*** *if for every problem $X \in NP$, $X \leq_P Y$.* ♣

Proposition 8.9

Suppose $Y \in NP\text{-}C$, then $Y \in P$ iff $P = NP$. ♠

Example 8.2 The first NP-C problem is Sat by Cook and Levin. Problems in the previous section, such as 3-Sat and Subset-Sum, are instances of NP-C problems.

Most NP problems are know to be either P or NP-C. Indeed, unless P = NP, there exists problems in NP that are neither P not NP-C (Ladner, 1975).