Theory of Algorithms

Laura Zielinski

Winter 2023

Stable Matching

See lecture notes for the Gale-Shapley algorithm for stable matching.

Theorem 1.3. *The algorithm always terminates in $O(n^2)$ steps and results in a stable matching.*

Theorem 1.8. *Assume people in group A make offers to group B. No matter what order the people in group A make their offers, the resulting matching will be the same. Moreover, this matching is the best possible stable matching for the people in group A and the worst possible stable matching for the people in group B. In other words, given a second stable matching, each person in group A prefers their first match and each person in group B prefers their second match.*

Big O Notation

Definition 1.1. *Given functions $f, g : \mathbb{N} \to \mathbb{R}^+$,*

1. *f is $O(g)$ if there exist $n_0, C > 0$ such that if $n \geq n_0$, we have $f(n) \leq C \cdot g(n)$.*
2. *f is $\Omega(g)$ if there exist $n_0, c > 0$ such that if $n \geq n_0$, we have $c \cdot g(n) \leq f(n)$.*
3. *f is $\Theta(g)$ if f is both $O(g)$ and $\Omega(g)$. In other words, f is $\Theta(g)$ if there exist $n_0, c, C > 0$ such that if $n \geq n_0$, we have $c \cdot g(n) \leq f(n) \leq C \cdot g(n)$.*

Proposition 1.3.

1. *Given functions $f, g, h : \mathbb{N} \to \mathbb{R}^+$, if f is $O(g)$ and g is $O(h)$, then f is $O(h)$.*
2. *Given functions $f_1, f_2, g_1, g_2 : \mathbb{N} \to \mathbb{R}^+$, if f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1 + f_2$ is $O(\max\{g_1, g_2\})$.*
3. *Given functions $f_1, f_2, g_1, g_2 : \mathbb{N} \to \mathbb{R}^+$, if f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1 f_2$ is $O(g_1 g_2)$.*

The corresponding statements for Ω and Θ are similar. Note that if f_1 is $O(g_1)$ and f_2 is $O(g_2)$, this does not imply that f_1/f_2 is $O(g_1/g_2)$. However, you can say that if f_1 is $\Theta(g_1)$ and f_2 is $\Theta(g_2)$, then f_1/f_2 is $\Theta(g_1/g_2)$.

Lemma 1.5. *Logarithms grow more slowly than any polynomial and exponential functions grow more quickly than any polynomial. In other words,*

1. *For all $C, c > 0$, there exists $n_0 > 0$ such that if $n \geq n_0$, then $C \cdot \log_2(n) \leq n^c$.*
2. *For all $C, c, c' > 0$, there exists $n_0 > 0$ such that if $n \geq n_0$, then $C \cdot n^c \leq 2^{c'n}$.*

Let $T(n)$ be the maximum number of steps an algorithm takes on an input of size n.

Example 1.6 (Polynomial Time). *If an algorithm takes $n^{O(1)}$ (polynomial) time, then there exist $n_0, C > 0$ such that if $n \geq n_0$, then $T(n) \leq n^C$.*

Example 1.7 (Exponential Time). *If an algorithm takes $2^{O(n)}$ (exponential) time, then there exist $n_0, C > 0$ such that if $n \geq n_0$, then $T(n) \leq 2^{Cn}$.*

Greedy Algorithms

Interval Scheduling

See lecture notes for the greedy algorithms for interval scheduling (Maximizing the Number of Jobs Accepted and Minimizing Maximum Lateness).

DFS and BFS

Algorithm 3.2 (Algorithms for Directed Connectivity). *This algorithm determines whether there is a path from one vertex s to another t, given a directed graph $G = (V, E)$.*

Keep track of a set S of vertices reachable from s, and a set E' of edges. Begin with $S = \{s\}$ and $E' = \{(s, v) \mid (s, v) \in E\}$. If E' is empty, return false. Remove an edge (u, v) from E', if we reach t return true, otherwise add v to S (if not already in S) and add all the edges starting at v to E'.

If E' is stored as a stack, then this is depth first search; if queue, then this is breadth first search.

Djikstra's Algorithm

Algorithm 4.2. *Dijkstra's algorithm finds a path of minimum length from s to t by iteratively finding the paths of minimum length from s to every other vertex in G.*

Theorem 4.3. *Dijkstra's algorithm finds a path from s to t of minimum length and can be implemented in $O(|E|log|E|)$ time.*

Lemma 4.4. *For all $v \in S$, we always have that d_v is the minimum length of a path from s to v.*

Priority Queues

A priority queue stores elements with their weights and supports insertion and deletion of the minimum value. One implementation is with a balanced binary tree where each node has a smaller weight than its children.

Minimum Spanning Trees

Definition 5.1. *Given an undirected graph $G = (V, E)$, we say that $T \subseteq E$ is a spanning tree of G if (V, T) is connected and contains no cycles.*

Problem 5.2. *The minimum spanning tree is a spanning tree T of G such that $\sum_{e \in T} l_e$ is minimized.*

Algorithm 5.3 (Kruskal's Algorithm). *Sort the edges in increasing order of their length. Consider the edges one by one and accept each edge which does not form a cycle with the accepted edges. Terminate when the accepted edges form a spanning tree.*

Algorithm 5.4 (Prim's Algorithm). *Start at a vertex s and iteratively accept the shortest edge which leads to a new vertex. Prim's algorithm works similarly to Dijkstra's algorithm.*

Definition 5.5. *Given a graph $G = (V, E)$ with all distinct edge lengths, define $E_{redundant}$ to be the set of edges e such that G contains a cycle C such that e is the longest edge in C.*

Theorem 5.6. *If all the lengths of the edges of G are distinct, there is a unique minimum spanning tree T of G with edges $E(T) = E(G) \setminus E_{redundant}$.*

Theorem 5.8. *Kruskal's algorithm and Prim's algorithm both give the minimum spanning tree.*

Union-Find

The Union-Find data structure has the following operations:

1. $Find(i)$: Return the representative r of the connected component containing i.

2. $Union(i, j)$: Merge the connected components containing vertices i and j and choose a new representative for this connected component.

We implement Union-Find by storing each connected component as a tree, where the root of the tree is the representative and every other vertex has a pointer to its parent. Thus, the operations are implemented as:

1. $Find(i)$: Starting from i, follow the pointer from each vertex to its parent until the root of the tree is reached.
2. $Union(i, j)$: After finding the roots of the trees containing i and j, merge the two trees by adding a pointer from one root to the other.

These two ideas can optimize Union-Find:

1. Merging by weight: When merging two connected components, the root of the merged tree should be the root of whichever tree was larger.
2. Path compression: Whenever we run $Find(i)$, we change the pointers of i and all the vertices along the path to point directly to the root.

Merging by Weight

Lemma 1.1. *When merging by weight, a tree with depth d has at least 2^d vertices.*

Thus, all trees have depth $O(\log n)$ so each operation is $O(\log n)$.

Path Compression

Lemma 1.5. *For any $Find(i)$ or $Union(i, j)$ operation, the sum of the number of steps needed and the change in the potential function is $O(\log_2 n)$.*

Divide and Conquer

To analyze the performance of recursive algorithms, let $T(n)$ be the time the algorithm takes on an input of size n and solve the recurrence relation.

Example 2.1 (mergesort). *The mergesort algorithm sorts an array of n elements as follows: if $n = 1$, the array is already sorted. Otherwise,*

1. *mergesort the first half of the array.*
2. *mergesort the second half of the array.*
3. *Merge the two sorted halves, which takes up to n steps.*

Up to a constant factor, the recurrence relation is $T(n) = 2T(n/2) + n$.

Example 2.2 (Binary search). *The binary search algorithm finds an element x in a sorted array as follows: take the middle element a_k of the array.*

1. *If $a_k = x$ we are done.*
2. *If $a_k < x$, then binary search the left half of the array.*
3. *If $a_k > x$, then binary search the right half of the array.*

Up to a constant factor, the recurrence relation is $T(n) = T(n/2) + 1$.

Solving Recurrence Relations

We can either expand out the recurrence relation, or take an educated guess. Expanding the recurrence relation may not work for complicated relations.

We consider recurrence relations of the form $L(T, n) = g(n)$ where $L(T, n)$ is linear in T. Thus, $L(T, n)$ is the homogeneous part and $g(n)$ is the inhomogeneous part. To find a general solution to $L(T, n) = g(n)$,

1. Find the vector space $\{T_0 \mid L(T_0, n) = 0\}$ of solutions to the homogeneous part.
2. Find a single solution T_1 to the entire relation $L(T_1, n) = g(n)$.

Then, $T = T_1 + T_0$ is a solution to the recurrence $L(T, n) = g(n)$.

For recurrence relations whose homogeneous part consists of the terms $T(cn)$ for some constant c, a good guess is $T_0(n) = n^p$ for some power p. If the terms instead are $T(n + c)$, try $T_0(n) = x^n$. A good guess for $T_1(n)$ is $c' \cdot g(n)$. If it fails, then try $T_1(n) = c' \log n \cdot g(n)$.

Theorem (Master theorem). *For the recurrence relation $T(n) = aT(n/b) + n^c$,*

1. *If $c = \log_b a$, then $T(n)$ is $O(n^c \log n)$.*
2. *If $c < \log_b a$, then $T(n)$ is $O(n^{\log_b a})$.*
3. *If $c > \log_b a$, then $T(n)$ is $O(n^c)$.*

Counting Inversions

We are given a sequence $a_1, \ldots, a_n$ of distinct numbers, and we are asked how many pairs $i < j \in [n]$ there are such that $i < j$ but $a_i > a_j$.

Algorithm 4.3. *Given an array A of n distinct elements, we can simultaneously count the number of inversions in A and sort A as follows:*

1. *If $n = 1$, the array is sorted and there are no inversions.*
2. *Apply this algorithm to the first half of A and let $count_L$ be its number of inversions.*
3. *Apply this algorithm to the second half of A and let $count_R$ be its number of inversions.*
4. *We can merge two sorted halves B and C as follows. If the first element of B is smaller, add it to the sorted sequence and remove it from B. If the first element of C is smaller, add it to the sorted sequence, remove it from C, and increase $count_{between}$ by $|B|$.*

After applying this algorithm to A, we have sorted A and the total number of inversions is $count_L + count_R + count_{between}$. The algorithm runs in $O(n \log n)$ time.

Closest Pair of Points

Given a set $P \subseteq \mathbb{R}^2$ of n points in the plane, which two points p_i and p_j in P are closest to each other?

Algorithm 5.2. *Given arrays X and Y of the points sorted by their x-coordinate and y-coordinate respectively,*

1. *If $n \leq 1$, the algorithm returns nothing. If $n = 2$, the algorithm returns the only pair of points.*
2. *Take the middle entry of X. Take P_L to be the points to the left and P_R to be the points to the right. Let X_L and Y_L be the sorted arrays containing the points in P_L and similarly for X_R and Y_R (this is done in $O(n)$ time).*
3. *Find the closest pairs of points in the left half and the right half.*
4. *Let d be the minimum distance found so far, let P_{mid} be the points that have x-coordinates less than d away from the center dividing line. Let Y_{mid} be the points in P_{mid} sorted by their y-coordinate. Consider all pairs p_i, p_j such that p_j is between 1 and 11 positions after p_i in Y_{mid}. Find the minimum distance.*

The algorithm runs in $O(n \log n)$ time.

Karatsuba Algorithm for Multiplication

Given two n-digit numbers x and y, taking $k = \lceil n/2 \rceil$, we can write $x = 10^k x_1 + x_2$ and $y = 10^k y_1 + y_2$, where x_1, y_1 are $(n-k)$-digit numbers and x_2, y_2 are k-digit numbers. Hence,

$$xy = 10^{2k} x_1 y_1 + 10^k x_1 y_2 + 10^k x_2 y_1 + x_2 y_2.$$

We only need to compute three products, $(x_1 + x_2)(y_1 + y_2)$, $x_1 y_1$, and $x_2 y_2$, as

$$xy = 10^{2k} x_1 y_1 + 10^k ((x_1 + x_2)(y_1 + y_2) - x_1 y_1 - x_2 y_2) + x_2 y_2.$$

(We do not count multiplying by 10^k, which is easy.) The recurrence relation is $T(n) = 3T(n/2) + O(n)$ (addition is $O(n)$), which gives $O(n^{\log_2 3})$.

Dynamic Programming

In dynamic programming, we solve problems by solving smaller subproblems then combining the solutions.

Weighted Interval Scheduling

Each job has a time interval $[a_i, b_i]$ and a weight w_i. We are asked to find a schedule S which maximizes the total weight of the accepted jobs. Sort $\{a_i \mid i \in [n]\}$ and $\{b_i \mid i \in [n]\}$ and let $(t_1, \ldots, t_{2n})$ be the sorted times. Define $w(t)$ to be the maximum total weight which can be completed by time t.

Lemma 2.5. *For all $k \in [2n-1]$, if $t_{k+1} = a_i$ then $w(t_{k+1}) = w(t_k)$. If $t_{k+1} = b_j$ for some $j \in [n]$ then $w(t_{k+1}) = \max\{w(t_k), w(a_j) + w_j\}$.*

The Bellman-Ford Algorithm for Shortest Paths

We can find the path from s to t of minimum length in a graph with negative edge weights but no negative cycles. Define $d_k(v)$ to be the minimum length of a walk from s to v which uses at most k edges.

Lemma 3.4. $d_{k+1}(v) = \min\{d_k(v), \min_{(u,v)\in E(G)}\{d_k(u) + l_{uv}\}\}$.

We have that $d_{|V(G)|}(t)$ is the path of minimum length. The algorithm takes $O(mn)$ time where $n = |V(G)|$ and $m = |E(G)|$. If there are any updates when $k = |V(G)|$, then the graph contains a negative cycle.

Knapsack

We are given n objects with weights w_i and values v_i, as well as a capacity C. We are asked for the maximum total value of objects we can carry. Define $v(k, w) = \max_{I \subseteq [k]} \sum_{i \in I} v_i$ where $\sum_{i \in I} w_i \leq w$.

Lemma 4.3. $v(k+1, w) = \max\{v(k, w), v(k, w - w_{k+1}) + v_{k+1}\}$.

Longest Common Subsequence

We are asked to find the longest common subsequence contained in two strings $a_1 a_2 \ldots a_n$ and $b_1 b_2 \ldots b_m$. Let $S(i, j)$ be the longest common subsequence in $a_1 a_2 \ldots a_i$ and $b_1 b_2 \ldots b_j$. If $a_i = b_j$, then $S(i, j) = S(i-1, j-1) + 1$. If $a_i \neq b_j$, then $S(i, j) = \max\{S(i-1, j), S(i, j-1)\}$.

Non-Crossing Matchings

We are asked to find the largest matching M of an undirected graph G with vertices $\{v_1, \ldots, v_n\}$ arranged in a circle, with no crossings. Define $M(i,j)$ to be the size of the largest non-crossing matching M such that M only contains edges between the vertices $v_i, \ldots, v_j$. We have that

$$M(i,j) = \max\{M(i,j-1), \max_{i \le i' < j, (v_{i'}, v_j) \in E(G)} M(i,i'-1) + M(i'+1, j-1) + 1\}.$$

There are $O(n^2)$ subproblems and it takes $O(n)$ time to compute each $M(i,j)$, so the total runtime is $O(n^3)$.

Maximum Flow

Given a directed graph where each edge has a capacity range $[a_e, b_e]$, what is the maximum possible flow from s to t? For all vertices other than s, t, the flow in must equal the flow out. The value of the flow is $f_{out}(s) - f_{in}(s) = f_{in}(t) - f_{out}(t)$.

Algorithm 2.4 (Ford-Fulkerson). *Iteratively, find a way to route flow from s to t in the residual graph, which describes the remaining capacities for the edges. Add this flow to the current flow, increasing the value of the flow from s to t. Update the residual graph accordingly.*

Definition 3.2. *A cut separating s and t is a partition $C = (L, R)$ of vertices such that $s \in L$ and $t \in R$. We define capacity$(C) = \sum_{(u,v)|u \in L, v \in R} b_e + \sum_{(u,v)|v \in L, u \in R}(-a_e)$.*

Corollary 3.6. *If f is a flow from s to t and C is a cut separating s and t, then value$(f) \le$ capacity(C). Thus, when Ford-Fulkerson terminates with flow f, there is a cut C with value$(f) =$ capacity(C). Take $C = (L, R)$ where $L = \{v \mid$ there is a flow path from s to v in the residual graph$\}$.*

Corollary 3.7 (Max flow/min cut). *If G has integer capacities, then the maximum flow from s to t is equal to the minimum capacity of a cut C separating s and t.*

Maximum Matching in Bipartite Graphs

We are given a bipartite graph G with vertices $A \cup B$ and edges $E(G) \subseteq \{(u,v) \mid u \in A, v \in B\}$. We are asked to find the largest matching $M \subseteq E(G)$ such that no two edges in M are incident to the same vertex. We can reduce maximum matching on bipartite graphs to max flow.

Add vertices s, t to G such that s has an edge to every vertex in A and every vertex in B has an edge to t. Give every edge capacity 1 and make edges point from A to B.

Proposition 4.2. *There is a bijection between integer valued flows $f : E(G') \to \{0,1\}$ from s to t on G' with value k and matchings M in G with k edges.*

Definition 4.3. *A set of vertices $V \subseteq V(G)$ is a vertex cover if for every edge $(u,v) \in E(G)$, either $u \in V$ or $v \in V$. Given a vertex cover V, the cut $L = \{s\} \cup (A \setminus V) \cup (V \cap B)$, $R = (A \cap V) \cup (B \setminus V) \cup \{t\}$ is a cut with capacity $|V|$. Given a cut (L, R) with capacity c, $V = (A \cap R) \cup (B \cap L)$ is a vertex cover of size c.*

Theorem 4.4 (König's theorem). *If G is a bipartite graph then the maximum size of a matching in G is equal to the minimum size of a vertex cover of G.*

Definition 4.5. *Given a matching M, we say that a path $P = v_0, v_1, \ldots, v_\ell$ is an augmenting path of edges in G if ℓ is odd, v_0 and v_ℓ are unmatched in M, and if i is even, then $(v_{i-1}, v_i) \in M$, and if i is odd, then $(v_{i-1}, v_i) \notin M$.*

Proposition 4.6. *If there is an augmenting path then we can increase the size of M by 1 by replacing edges (v_{i-1}, v_i) (i is even) with the edges (v_{i-1}, v_i) (i is odd).*

Lemma 4.7. *If M is not a maximum matching of G then there exists an augmenting path.*

Menger's Theorem

To find all edge-disjoint paths from s to t in G, find a maximum flow from s to t when all edges have capacity 1.

To find all vertex-disjoint paths from s to t in G, split each vertex $v \in V(G)$ into v_{in} and v_{out} with an edge between them. Replace each edge (u, v) with an edge (u_{out}, v_{in}). Then find the maximum flow from s to t.

Definition 5.1. *S is a vertex separator separating s and t if $S \subseteq V(G) \setminus \{s, t\}$ and every path from s to t has a vertex in S.*

Theorem 5.2 (Menger's theorem). *The number of vertex-disjoint paths from s to t is equal to the minimum size of a vertex separator separating s and t.*

NP Completeness

Definition 1.1. *A YES/NO problem is in P if there is a polynomial time algorithm to solve the problem.*

Definition 1.7. *A non-deterministic polynomial time algorithm for a YES/NO problem P is an algorithm A such that:*

1. *If the answer to P is NO, A always returns NO.*
2. *If the answer to P is YES, the probability A returns YES is positive.*
3. *There exist $B, c > 0$ such that A terminates in time Bn^c (A takes polynomial time).*

Definition 1.8. *A YES/NO problem is in NP if there is a non-deterministic polynomial time algorithm for it.*

Definition 1.11. *A polynomial time verifier for a YES/NO problem is an algorithm V such that:*

1. *The input to V is the input to the problem $x \in \{0,1\}^n$ and a potential solution $s \in \{0,1\}^m$, where m is polynomial in n.*
2. *If the answer to the problem is YES, there exists a solution s such that $V(x, s) = 1$. If the answer to the problem is NO, then for all solutions s, we have $V(x, s) = 0$.*
3. *V takes polynomial time.*

Definition 1.12. *A YES/NO problem is in NP if there is a polynomial time verifier for it.*

Lemma 1.13. *There is a non-deterministic polynomial time algorithm for a problem if and only if there is a polynomial time verifier for it.*

Reductions

Definition 2.1. *A problem A is poly-time reducible to problem B if there is a polynomial time algorithm which takes an instance x of problem A of size n and returns an instance y of problem B of size $poly(n)$ such that $A(x) = B(y)$.*

Remark 2.1. *If we can reduce problem A to problem B and we can solve problem B, then we can solve problem A. If problem A is hard, then problem B is hard. If problem B is easy, then problem A is easy.*

If problem A and problem B are reducible to each other, then they are equivalent.

Proposition 2.2. *If problem A is poly-time reducible to problem B and problem B is poly-time reducible to problem C, then problem A is poly-time reducible to problem C.*

NP Hardness and NP Completeness

Definition 3.1. *A problem P is NP-hard if every problem P' in NP is poly-time reducible to P.*

Definition 3.3. *A problem P is NP-complete if P is both NP and NP-hard.*

Remark 3.3. *To show a problem P is NP-complete, we must show P is in NP (give a poly-time verifier or poly-time non-deterministic algorithm), then give a poly-time reduction from an NP-complete problem to P. To prove a reduction is correct, show that:*

1. *Given a solution to the original instance of P, you can find a solution to the instance you constructed.*
2. *Given a solution to the instance you constructed, you can find a solution to the original instance of P.*

Proposition 3.4. *If P and P' are both NP-complete, then P and P' are poly-time reducible to each other.*

Important NP-Complete Problems

Problem (Independent Set). *Given a graph G, is there an independent set of size k in G? An independent set is a set of vertices $I \subseteq V(G)$ such that no two vertices in I are adjacent to each other.*

Problem (Vertex Cover). *Given a graph G, is there a vertex cover of size k in G? A vertex cover is a set of vertices $V \subseteq V(G)$ such that for every edge $(u, v) \in E(G)$, either $u \in V$ or $v \in V$.*

Problem (Clique). *Given a graph G, can you find a $k-$clique? A clique is a set of vertices that are each adjacent to all the others.*

Theorem 2.5. *Independent Set, Vertex Cover, and Clique are poly-time reducible to each other.*

Remark 2.5. *Independent Set and Clique can be reduced to each other by taking the complement of the graph.*

Lemma 2.6. *There is an independent set of size k in G if and only if there is a vertex cover of size $n - k$ in G, by taking the complement of each set.*

Problem (Circuit-SAT). *Given a boolean circuit C with one output, is there a set of inputs which makes C output 1?*

Theorem (Cook-Levin). *Circuit-SAT is NP-complete.*

Problem (3-SAT). *Given a boolean expression in CNF where each clause has at most 3 literals (x_i or $\neg x_i$), is there a set of inputs to make it output 1?*

Problem (3-Coloring). *Given a graph G, can you color each vertex with one of 3 different colors so that no two adjacent vertices have the same color?*

Problem (Knapsack). *Given non-negative weights $w_1, \ldots, w_n$, values $v_1, \ldots, v_n$, capacity C, and minimum value V, is there a subset $S \subseteq [n]$ such that $\sum_{i \in S} w_i \leq C$ and $\sum_{i \in S} v_i \geq V$?*

Problem (Hamiltonian Path/Cycle). *Given a graph G, is there a path/cycle which visits each vertex exactly once?*