1 Regular languages

- Finite automaton: $(Q, \Sigma, \delta, q_0, F)$, where $\delta : Q \times \Sigma \to Q$, $q_0 \in Q$, $F \subseteq Q$
- M accepts w if sequence of states $r_0, \ldots, r_n$ exists: $r_0 = q_0$, $\delta(r_i, w_{i+1}) = r_{i+1}$ for i = 0 to $n - 1$, $r_n \in F$
- Regular languages are closed under the operations union, concatenation and star
- Nondeterministic finite automaton: $(Q, \Sigma, \delta, q_0, F)$, where $\delta : Q \times \Sigma_\epsilon \to P(Q)$, $q_0 \in Q$, $F \subseteq Q$
- Every NFA has an equivalent DFA: $Q' = P(Q)$, $\delta'(R, a) = \bigcup_{r \in R} E(\delta(r, a))$, $q_0' = \{E(q_0)\}$, $F' = \{R \in Q' | R \text{ contains an accept state of } N\}$
- Regular expression: $a \in \Sigma$, ϵ, $\emptyset$, $(R_1 \cup R_2)$, $(R_1 \circ R_2)$, (R_1^*)
- Convert a DFA into a regular expression:
  - GNFA: NFA where transition arrows are regular expressions; start state has transition arrows going to every other state but no arrows coming in from any other state; only one accept state with arrows coming in from every other state but no arrows going to any other state, accept state is not the start state; one arrow goes from every state to every other state and also from each state to itself; $\delta : (Q - \{q_{\text{accept}}\}) \times (Q - \{q_{\text{start}}\}) \to R$
  - $(R_1)(R_2)^*(R_3) \cup (R_4)$
- Pumping lemma: A is a regular language, $s \in A$ with length at least p, $s = xyz$:
  1. For each $i \geq 0$, $xy^iz \in A$
  2. $|y| > 0$
  3. $|xy| \leq p$

2 Context-free languages

- Context-free grammar: (V, Σ, R, S): V = variables, Σ = terminals (disjoint from V), R = rules (a rule being a variable and a string of variables and terminals), $S \in V$ is start variable
- Convert DFA into CFG: Make a variable R_i for each state q_i. Add rule $R_i \to aR_j$ to the CFG if $\delta(q_i, a) = q_j$ is a transition. Add rule $R_i \to \epsilon$ if q_i is an accept state of the DFA. R_0 is the start variable of the grammar.
- A leftmost derivation is a derivation where the leftmost variable is replaced. A string is derived ambiguously if it has two or more different leftmost derivations in a grammar.
- Chomsky normal form: $A \to BC$, $A \to a$
- Conversion to CNF: Add a new start variable S_0 and the rule $S_0 \to S$. Eliminate all ϵ rules not involving the start variable. Remove unit rules: $A \to B$ and $B \to u$, write $A \to u$. Convert into proper form, and replace terminals.
- Pushdown automaton: $(Q, \Sigma, \Gamma, \delta, q_0, F)$: Γ = stack alphabet, $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to P(Q \times \Gamma_\epsilon)$, $q_0 \in Q$, $F \subseteq Q$:
  1. $r_0 = q_0$, $s_0 = \epsilon$
  2. $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $s_i = at$, $s_{i+1} = bt$
  3. $r_m \in F$
- Convert a CFG to a PDA:
  1. Place \$ and start variable on stack

2. Repeat forever:
   (a) If top of stack is variable A, nondeterministically select one of the rules for A and substitute A by string on the right-hand side of the rule
   (b) If top of stack is terminal a, read next symbol from input and compare to a. If they match, repeat, otherwise, reject.
   (c) If top of stack is \$, enter accept state.

- Convert a PDA to a CFG:
  1. Modify P to have the following features: it has a single accept state, it empties the stack before accepting, each transition pushes or pops but does not do both
  2. For each pair of states p and q, we have a variable A_{pq}. If the stack is only ever empty at the beginning and end, $A_{pq} \to aA_{rs}b$. If the stack is empty in between, $A_{pq} \to A_{pr}Arq$.
  3. $A_{pp} \to \epsilon$
- Pumping lemma: A is a CFL, $s \in A$ with length at least p, $s = uvxyz$:
  1. For each $i \geq 0$, $uv^ixy^iz \in A$
  2. $|vy| > 0$
  3. $|vxy| \leq p$
- CFL is closed under union, concatenation and star

3 Church-Turing thesis

- Turing machine: $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$: Σ = input alphabet not including blank, Γ = tape alphabet (has a blank, $\Sigma \subseteq \Gamma$, $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$, $q_0 \in Q$, $q_{\text{reject}} \neq q_{\text{accept}}$
- uaq_ibv yields uq_jacv if $\delta(q_i, b) = (q_j, c, L)$
- uaq_ibv yields $uacq_jv$ if $\delta(q_i, b) = (q_j, c, R)$
- Multi-tape Turing machine: $\delta Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k$
- Nondeterministic Turing machine: $\delta : Q \times \Gamma \to P(Q \times \Gamma \times \{L, R\})$, can be simulated by a DTM by having an address tape that tells us which children to follow
- Turing-recognizable iff some enumerator enumerates it. Convert enumerator to TM: run E and compare it to w. Convert TM to enumerator: If $s_1, \ldots$ is a list of all possible strings in Σ^*, for all i, run M for i steps on $s_1, \ldots, s_i$, if any computations accept, print out s_j.
- Church-Turing thesis: connection between informal notion of algorithm and precise definition, Turing machines and λ-calculus are equivalent

4 Decidability

- $A_{DFA} = \{\langle B, w\rangle | B \text{ is a DFA that accepts input string } w\}$ is decidable: just simulate w on B
- A_{NFA} and A_{REX} are decidable
- E_{DFA} is decidable: a graph search for the accept states
- EQ_{DFA} is decidable: $L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right)$
- A_{CFG} is decidable: convert to Chomsky normal form, list all derivations with $2n - 1$ steps, except if $n = 0$, where you take 1 step

- E_{CFG} is decidable: mark all terminals, mark variables where each symbol has been marked
- Every context-free language is decidable: use A_{CFG}
- A_{TM} is undecidable: acceptance problem. $H(\langle M, w\rangle)$ accepts if M accepts W, reject otherwise. $D(\langle M\rangle)$: run H on $\langle M, \angle M\rangle\rangle$, output the opposite of what H outputs. Run $D(\langle D\rangle)$.
- Some languages are not Turing-recognizable: the set of all Turing machines is countable, where as the set of all infinite binary sequences is uncountable (take the characteristic sequences)
- A language is decidable iff it is Turing-recognizable and co-Turing-recognizable.
- $\overline{A_{TM}}$ is not Turing-recognizable.

5 Reducibility

- Halting problem: HALT is undecidable: show A is reducible to HALT
- E is undecidable: create a machine that runs M only if $x = w$
- REGULAR is undecidable: create a machine that accepts 0^n1^n, and accepts all other strings if M accepts w
- EQ is undecidable: show E is reducible to EQ by equating M with a machine that always rejects
- A_{LBA} is decidable: run for qng^n steps, q states, g symbols, tape of length n
- E_{LBA} is undecidable: reduction from A
- ALL for CFG (G is a CFG and $L(G) = \Sigma^*$) is undecidable
- PCP is undecidable:
  - MPCP starts with first domino
  - First domino is $\left[\frac{\#}{\#q_0w_1w_2\cdots w_n\#}\right]$
  - For every $a, b \in \Gamma$ and every $q \neq q_{\text{reject}}, r \in Q$, if $\delta(q, a) = (r, b, R)$, have $\left[\frac{qa}{br}\right]$
  - For every $a, b, c \in \Gamma$ and every $q \neq q_{\text{reject}}, r \in Q$, if $\delta(q, a) = (r, b, L)$, have $\left[\frac{cqa}{rcb}\right]$
  - For every $a \in \Gamma$, have $\left[\frac{a}{a}\right]$
  - Have $\left[\frac{\#}{\#}\right]$ and $\left[\frac{\#}{\sqcup\#}\right]$
  - For every $a \in \Gamma$, have $\left[\frac{aq_{\text{accept}}}{q_{\text{accept}}}\right]$ and $\left[\frac{q_{\text{accept}}a}{q_{\text{accept}}}\right]$
  - Add $\left[\frac{q_{\text{accept}}\#\#}{\#}\right]$
- Computable function: some Turing machine halts with just $f(w)$ given w as input
- Language A is mapping reducible to language B, written $A \leq_m B$, if there is a computable function f, where for every w, $w \in A \Leftrightarrow f(w) \in B$. f is the reduction of A to B.
- If $A \leq_m B$ and B is decidable, then A is decidable.
- If $A \leq_m B$ and A is undecidable, then B is undecidable.
- If $A \leq_m B$ and B is Turing-recognizable, then A is Turing-recognizable.
- If $A \leq_m B$ and A is not Turing-recognizable, then B is not Turing-recognizable.
- EQ_{TM} is neither Turing-recognizable nor co-Turing-recognizable.

- Rice's theorem: The problem of determining whether a given Turing machine's language has a non-trivial property P is undecidable. The property P is such that some TMs have it and others do not, and if two TMs recognise the same language, either both have P or neither do.

6 Time complexity

- Every context-free language is in P
- Verifier is an algorithm V: $A = \{w | V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}$
- NP is the class of languages that have polynomial time verifiers. A language is in NP iff it is decided by some nondeterministic polynomial Turing machine
- NTIME($t(n)$) = set of languages decided by $O(t(n))$ time NTM
- NP-complete problems are those problems where if a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solveable.
  1. B is in NP, and
  2. every A in NP is polynomial time reducible to B
- Cook-Levin theorem: SAT in P iff P = NP, or SAT is NP-complete
- Polynomial time mapping reducible: $A \leq_p B$ iff $w \in A \Leftrightarrow f(w) \in B$
- If $A \leq_p B$ and $B \in P$, then $A \in P$.
- 3SAT is the problem is deciding a boolean clause in conjunctive normal form (AND of ORs) and every clause has three literals. 3SAT is polynomial time reducible to CLIQUE.
- If B is NP-complete and $B \in P$, then P = NP
- If B is NP-complete and $B \leq_p C$ for C in NP, then C is NP-complete

7 Load balancing problem

- Greedy algorithm: assign to machine with least amount of load
- $T^* \geq \frac{1}{n} \sum t_j$, and $T^* \geq \max t_j$
- $T^* \leq T \leq 2T^*$
- Proof of 2-approximation:
  - Let t_j be the last job assigned to M_i. Load assigned to M_i before t_i is $T_i - t_j$.
  - $\sum T_k \geq m(T_i - t_j)$
  - $T_i - t_j \leq \frac{1}{m} \sum T_k = \frac{1}{m} \sum t_j \leq T^*$
  - $T_i - t_j + t_j \leq T^* + t_j$
- For 1.5-approximation, take the jobs in decreasing order of size:
  - If there are more than m jobs, $T^* \geq 2t_{m+1}$, so $t_j \leq t_{m+1} \leq \frac{T^*}{2}$.

8 Set cover

1. Start with $R = U$ and no sets selected (R is like the remainder).
2. While $R \neq \emptyset$:
   (a) Select set S_i which minimises:
   $$\frac{w_i}{|S_i \cap R|}$$
   (b) Define:
   $$c_i = \frac{w_i}{|S_i \cap R|}$$
   $\forall s \in S_i \cap R$.
   (c) Delete set S_i from R.
3. Return selected sets.

- If C is the set cover obtained by the greedy set cover algorithm, then:
$$\sum_{S_i \in C} w_i = \sum_{s \in U} c_s$$
- Claim: For every set S_k, $\sum_{s \in S_k} c_s$ is at most $H(|S_k|)w_k$.
- Proof:
  - Assume the elements of S_k are the first d elements of U: $S_k = \{s_1, s_2, \ldots, s_d\}$.
  - Assume that the elements are labelled in the order in which they are assigned c_{s_j} by the algorithm.
  - Consider the iteration in which s_j, $j \leq d$ is picked up.
  - At the start of the iteration, $s_j, s_{j+1}, \ldots, s_d \in R$.
  - $|S_k \cap R| \geq d - j + 1$
  - If it picks S_i (greedy algorithm comes into play here):
  $$c_{s_j} = \frac{w_i}{|S_i \cap R|} \leq \frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}$$
  $$\sum_{s \in S_k} c_s = \sum_{j=1}^{d} c_{s_j} \leq \sum_{j=1}^{d} \frac{w_k}{d - j + 1} = \frac{w_k}{d} + \frac{w_k}{d-1} + \cdots + \frac{w_k}{1}$$
  - $w_k H(d) = w_k \left(\frac{1}{d} + \frac{1}{d-1} + \cdots + 1\right)$
- Claim: The set cover C selected by the greedy algorithm has weight at most $H(d^*)w^*$, where w^* is the optimal weight.

9 Pricing method

- Vertex cover problem:
  - Given $G(V, E)$, a **vertex cover** is a set $S \subset V$ such that each edge is incident on at least one element of S.
  - A weight w_i is associated with each vertex $v_i \in V$.

- The objective is to minimise the weight of the vertex cover:

$$W(S) = \sum_{i \in S} w_i$$

- There is a mapping from vertex cover to the set cover problem. The edges become the set U to be covered, all the edges incident on each vertex become a subset S_i of U, and the weights of the vertices become the weights of S_i.
- Previously, the approximation algorithm for set cover had a constant factor of $H(d^*)$, where d^* was the maximum size of the sets. We now present a 2-approximation algorithm.

- Approximation algorithm for the vertex cover problem:
  - We want to associate a "price" with every edge e, $p_e \geq 0$.
    Definition: A price p_e is **fair** if for each vertex i:

$$\sum_{e=(i,j)} p_e \leq w_i$$

  - Claim: For any vertex cover S^* and any non-negative and fair price p_e:

$$\sum_{e \in E} p_e \leq w(S^*)$$

    Proof:

$$\sum_{e=(i,j)} p_e \leq w_i \, \forall i \in S^*$$

$$\sum_e p_e \leq \sum_{i \in S^*} \sum_{e=(i,j)} p_e \leq \sum_{i \in S^*} w_i = w(S^*)$$

  - Definition: A vertex i is **tight** if:

$$\sum_{e=(i,j)} p_e = w_i$$

  - VERTEX-COVER-APPROX(G, w):
    1. Set $p_e = 0 \, \forall e \in E$
    2. While there is an $e = (i, j)$ such that neither i or j is tight:
       (a) Select such an e.
       (b) Increase p_e without violating fairness.
    3. Let S be the set of all tight vertices.
    4. Return S.
  - Claim: The set S and prices returned by the algorithm satisfy:

$$w(S) \leq 2 \sum_{e \in E} p_e$$

    Proof: All nodes in S are tight.

$$\sum_{e=(i,j)} p_e = w_i \, \forall i \in S$$

$$w(S) = \sum_{i \in S} w_i = \sum_{i \in S} \sum_{e=(i,j)} p_e \leq 2 \sum_{e \in E} p_e \leq 2w(S^*)$$

    Thus, we have a 2-approximation algorithm.
  - If you have a greedy algorithm that choose vertices in order of degree, you can have an approximation that is arbitrarily large.

10 Linear programming

- Standard form:
  - Minimise $c^T \cdot x$
  - Subject to $A \cdot x \geq b$, and $x \geq 0$
- Integer programming problem: related to linear programming, but in addition to having $x \geq 0$, we have the constraint that x is an integer. This is an NP-hard problem.

10.1 Reducing vertex cover to integer programming

- Let $x_i = 1$ if $i \in S$, $x_i = 0$ otherwise.
- Minimise $\sum_{i \in V} w_i x_i$
- $x_i + x_j \geq 1 \, \forall \, e = (i, j) \in E$
- $x_i \in \{0, 1\} \, \forall \, i \in V$
- Typically, in a graph problem, the vertices become the x_i, choose the objective function and then the problem will give the constraints.
- Approximate the integer programming problem with a linear programming problem. Remove the constraint of $\{0, 1\}$. $w_{\text{LP}} \leq w_{\text{IP}}$, and thus we have a lower bound for the solution to the integer programming problem.
- To map the problem back, do rounding, i.e. $S = \{i \,|\, x_i \geq 0.5\}$. You still have a feasible solution, because if $x_i, x_j < 0.5$, then $x_i + x_j < 1$, which violates the constraint, and so you never lose vertices that should have been selected.
- $w_{\text{LP}} w \cdot x^* = \sum_i w_i x_i^* \geq \sum_{i \in S} w_i x_i^* \geq \frac{1}{2} \sum_{i \in S} w_i = \frac{1}{2} w(S)$
- $w(S) \leq 2 w_{\text{IP}}$, and thus we have a 2-approximation algorithm.

10.2 Primal dual method

- Example: suppose min $4x_1 + 5x_2 + 7x_3$ subject to $x_1 + x_2 + x_3 \geq 2$ (1) and $x_1 + 2x_2 + 3x_3 \geq 8$ (2), $x_1, x_2, x_3 \geq 0$. Can I get a bound on this objective function?
- Multiply (1) by 4, and we get $4x_1 + 4x_2 + 4x_3 \geq 8$. The objective function is clearly bigger: $4x_1 + 5x_2 + 7x_3 \geq 8$.
- Multiply (2) by 2, and we get $4x_1 + 5x_2 + 7x_3 \geq 2x_1 + 4x_2 + 6x_3 \geq 16$.
- Take y_i multiples of equation number i, and then add them up. We get: $x_1(y_1 + y_2) + x_2(y_1 + 2y_2) + x_3(y_1 + 3y_2) \geq 2y_1 + 8y_2$. Comparing coefficients with the objective function, $y_1 + y_2 \leq 4$, $y_1 + 2y_2 \leq 5$, $y_1 + 3y_2 \leq 7$; maximise $2y_1 + 8y_2$; $y_1, y_2 \geq 0$.
- Dual problem: Maximise $b \cdot y$, subject to $A^T \cdot y \leq c^T$.
- The optimal value of the primal problem is the optimal value of the dual problem.
- The dual problem of the vertex cover problem: maximise $\sum_{(i,j) \in E} y_{ij}$, subject to $\sum_{j:(i,j) \in E} y_{ij} \leq w_i \, \forall \, i \in V$, $y_{ij} \geq 0$
- The node is tight if the constraints become equalities, and tight nodes turn out to be the vertex cover.

11 Weighted interval scheduling with dynamic programming

11.1 Problem description

- Given requests $\{1, \ldots, n\}$, each request has start time s_i, finish time f_i and value v_i.
- Find a set $S \subseteq \{1, \ldots, n\}$ of compatible intervals that maximises $\sum_{i \in S} v_i$.

11.2 Analysis

- Order: $f_1 \leq f_2 \leq \ldots \leq f_n$, that is $i \leq j \Leftrightarrow f_i \leq f_j$.
- $p(j)$ is the largest index $i < j$ such that i, j are disjoint.
- Let O be the optimal solution.
- If $n \in O$, then $\{p(n) + 1, \ldots, n - 1\} \nsubseteq O$. We can then split the problem up into a subproblem. The optimal solution of a subproblem forms part of the solution to the problem: WIS$(\{1, \ldots, p(n)\}) \subseteq O$.
- If $n \notin O$, then $O =$ WIS$(\{1, \ldots, n - 1\})$.
- $O_j =$ WIS$(\{1, \ldots, j\})$ and OPT$(j) = \sum_{i \in O_j} v_i$
- OPT$(j) = \max\big(v_j +$ OPT$(p(j)),$ OPT$(j - 1)\big)$: there are only two choices, that is, whether j is in the optimal solution or not. In other words, $j \in O_j$ iff $v_j +$ OPT$(p(j)) \geq$ OPT$(j - 1)$.
- We also need to calculate the solution S in addition to its value. We use the fact that $j \in O_j$ iff $v_j +$ OPT$(p(j)) \geq$ OPT$(j - 1)$ to extract S out of the algorithm by reconstructing the choices in the algorithm; there is no need to continually recalculate S.
- There are only a polynomial number of subproblems.