COMP 2907 - ALGORITHMS AND COMPLEXITY
EXAM NOTES

ANDREW TULLOCH AND GILES GARDAM

1. Introduction

Algorithm 1.1 Propose-and-reject algorithm for the stable matching problem

```
1: procedure StableMatching(M, W)
2:     while some man is free and hasn't proposed to every woman do
3:         Choose such a man m
4:         w ← first woman on m's list to whom m has not yet proposed
5:         if w is free then
6:             assign m and w to be engaged
7:         else if w preferms m to her fiance m' then
8:             assign m and w to be engaged, and m' to be free
9:         else
10:            w rejects m
11:        end if
12:    end while
13: end procedure
```

Theorem 1.1. *The Gale-Shapley algorithm yields a stable matching in $\mathcal{O}(n^2)$ time. The matching it returns is man-optimal (all men simultaneously get matched to the best possible woman over all stable matchings), and is hence independent of the order in which free men are chosen to propose.*

Definition 1.2 (Efficieny)**.** An algorithm is *efficient* if its worst-case running time is polynomial.

Definition 1.3 ($\mathcal{O}$, Θ, Ω)**.** We have the following bounds on the running time of an algorithm $T(n)$.

- $T(n)$ is $\mathcal{O}(f(n))$ if there exists $c > 0$ such that $T(n) \leq cf(n)$ for n sufficiently large.
- $T(n)$ is $\Omega f(n)$ if there exists $c > 0$ such that $T(n) \geq cf(n)$ for n sufficiently large.
- $T(n)$ is $\Theta(f(n))$ if it is both $\mathcal{O}(f(n))$ and $\Omega(f(n))$.

1.1. **Graphs.**

Definition 1.4 (Adjacency list)**.** An adjacency list is a node indexed array of lists. It contains two representations of each edge. Checking if (u, v) is an edge takes $\mathcal{O}(\deg u)$ time, identifying all edges and space used are both $\mathcal{O}(n+m)$. (Convention is that a graph has n vertices and m edges.)

Theorem 1.5 (Trees)**.** *An undirected graph is a* ***tree*** *if it is connected and does not contain a cycle.*

Let G be an undirected graph on n nodes. Any two of the following statements imply the third.

- *G is connected.*
- *G does not contain a cycle.*
- *G has $n-1$ edges.*

Theorem 1.6 (BFS algorithm)**.** *Explore outward from s in all possible directions, adding nodes one "layer" at a time.*

- $L_0 = \{s\}$
- L_i = *all nodes that do not belong to an earlier layer, and that have an edge to a node in* L_{i-1}.

Theorem 1.7. BFS *runs in $\mathcal{O}(m+n)$ time if the graph is given by its adjacency representation.*

Definition 1.8 (Bipartite graph)**.** A graph is bipartite if the nodes can be coloured red and blue such that every edge has one red and one blue end.

Theorem 1.9. *A graph is bipartite if and only if it does not contain an odd length cycle.*

Definition 1.10 (Strongly connected)**.** A (directed) graph is strongly connected if for every pair of nodes (u, v), there is a path from u to v and a path from v to u. Can be tested by BFS in $\mathcal{O}(m+n)$ time (exanding along 'forward' edges in one pass and then along 'backward' edges in the other).

Definition 1.11 (Directed acyclic graph)**.** A directed acyclic graph is a directed graph that contains no directed cycles.

Lemma 1.12. *A topological order is total ordering such that if an edge joins u to v then $u < v$. G has a topological order if and only if G is a directed acyclic graph.*

2. Greedy Algorithm

Algorithm 2.1 Greedy algorithm for interval scheduling **P**

```
1: procedure IntervalScheduling(J)
2:     Sort jobs by finish times so that f_1 ≤ f_2 ≤ ··· ≤ f_n.
3:     A ← ∅
4:     for j = 1 to n do
5:         if job j compatible with A then
6:             A ← A ∪ {j}
7:         end if
8:     end for
9:     return A
10: end procedure
```

2.1. **Interval scheduling.**

Theorem 2.1. IntervalScheduling *runs in* $\mathcal{O}(n \log n)$ *time.*

Algorithm 2.2 Greedy algorithm for interval partitioning

```
1: procedure IntervalPartitioning(J)
2:     Sort intervals by starting times so that s_1 ≤ s_2 ≤ ··· ≤ s_n.
3:     d ← 0
4:     for j = 1 to n do
5:         if lecture j compatible with some classroom k then
6:             schedule lecture j in classroom k
7:         else
8:             allocate a new classroom d + 1
9:             schedule lecture j in classroom d + 1
10:            d ← d + 1
11:        end if
12:    end for
13: end procedure
```

2.2. **Interval Partitioning.**

2.3. **Some simple greedy algorithms.**

- For jobs with fixed deadlines and durations to be scheduled on one machine, we can minimize the maximum lateness going by earliest deadline first.
- For optimal caching, we evict the item that is requested farthest in the future.

- To minimize the number of fuel stops, the truck driver's algorithm chooses the furthest possible stop at each step.
- To find the best k-clustering, i.e. partition of nodes into k sets which maximizes the minimum distance between nodes in different clusters, we can apply Kruskal's algorithm, ending when there are k connected compnents.

2.4. **Dijkstra's algorithm.** Given a directed graph $G = (V, E)$ with non-negative edge weights, finds the shortest path from a node s to a target node t (or any arbitrary node).

- Maintain a set of **explored nodes** S for which we have determined the shortest path distance $d(u)$ from s to u.
- Initialise $S = \{s\}, d(s) = 0$.
- Repeatedly choose unexplored node v which minimises

$$\pi(v) = \min_{(u,v):u\in S} d(u) + w_{uv},$$

  add v to S, and set $d(v) = \pi(v)$.

Theorem 2.2. DIJKSTRA *runs in* $\mathcal{O}(m \log n)$ *with a binary heap.*

2.5. **Minimum spanning trees.**

Definition 2.3 (Minimum spanning tree)**.** Given a connected graph $G = (V, E)$ with real-valued edge weights w_{uv}, an MST is a subset of the edges $T \subseteq E$ such that T is a spanning tree whose sum of edge weights is minimised.

Theorem 2.4 (Kruskal's algorithm)**.** *Start with $T = \emptyset$. Consider edges in ascending order of cost. Insert edge e in T unless doing so would create a cycle.*

Theorem 2.5 (Reverse-Delete algorithm)**.** *Start with $T = E$. Consider edges in descending order of cost. Delete edge e from T unless doing so would disconnect T.*

Theorem 2.6 (Prim's algorithm)**.** *Start with some root node s and greedily grow a tree from s outward. At each step, add the cheapest edge e to T that has exactly one endpoint in T.*

Proposition 2.7 (Cut property)**.** *Let S be any subset of nodes, and let e be the minimum cost edge with exactly one endpoint in S. Then the* MST *contains e.*

Proof. Assume the contrary. Adding e creates a cycle. Some other edge on the cut is more expensive than e, and removing it leaves a cheaper spanning tree. □

Proposition 2.8 (Cycle property)**.** *Let C be any cycle, and let f be the maximum cost edge belonging to C. Then the* MST *does not contain f.*

Proof. Assume the contrary. Removing f leaves two connected components, with one end of f in each. Some other edge in the cycle is on the cut, and adding it gives us a cheaper spanning tree. □

Algorithm 2.3 Prim's algorithm for minimal spanning tree

```
1: procedure PRIM(G, c)
2:     for all v ∈ V do a[v] ← ∞
3:     end for
4:     Initialise an empty priority queue Q
5:     for all v ∈ V do Insert v onto Q
6:     end for
7:     Initialise a set of explored nodes S ← φ
8:     while Q is not empty do
9:         u ← delete minimum element from Q.
10:        S ← S ∪ {u}
11:        for all edges e = (u, v) incident to u do
12:            if v ∉ S and c_e < a[v] then
13:                decrease priority a[v] to c_e
14:            end if
15:        end for
16:    end while
17: end procedure
```

Theorem 2.9. PRIM *runs in* $\mathcal{O}(n^2)$ *with an array,* $\mathcal{O}(m \log n)$ *with a binary heap.*

Algorithm 2.4 Kruskal's algorithm for minimal spanning tree

```
1: procedure KRUSKAL(G, c)
2:     Sort edge weights so that c_1 ≤ c_2 ≤ ··· ≤ c_m.
3:     T ← φ
4:     for all u ∈ V do make a set containing singleton u.
5:     end for
6:     for i = 1 to m do
7:         (u, v) = e_i
8:         if u and v are in different sets then
9:             T ← T ∪ {e_i}
10:            merge the sets containing u and v
11:        end if
12:    end for
13:    return T
14: end procedure
```

3. Divide & Conquer

Theorem 3.1 (Master theorem)**.** *If* $T(n) \leq aT(n/b) + \mathcal{O}(n^d)$*, then*

$$T(n) = \begin{cases} \mathcal{O}(n^d) & \text{if } a < b^d \\ \mathcal{O}(n^d \log n) & \text{if } a = b^d \\ \mathcal{O}(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Algorithm 3.1 Counting inversions

```
1: procedure SORTANDCOUNT(L)
2:     if list L has one element then
3:         return 0 and the list L
4:     end if
5:     Divide L into two halves A and B
6:     (r_A, A) ← SORTANDCOUNT(A)
7:     (r_B, A) ← SORTANDCOUNT(B)
8:     (r, L) ← MERGEANDCOUNT(A, B)
9:
10:    return r = r_A + r_B + r and the sorted list L
11: end procedure
```

Algorithm 3.2 Finding the closest pair of points in a plane

```
1: procedure ClosestPair(L)
2:     Compute separation line X such that half the points are on one side and half on the other side.
3:
4:     δ1 = ClosestPair(left half)
5:     δ2 = ClosestPair(right half)
6:     δ = min(δ1, δ2)
7:
8:     Delete all points further than δ from separation line X
9:     Sort remaining points by y-coordinate
10:    Scan points in y order and compare distance between each point and next 11 neighbours. If any of these distances is less than δ, update δ
11:
12:    return δ
13: end procedure
```

3.1. **Closest pair of points.**

Theorem 3.2. *Running time is* $T(n) \leq 2T(n/2) + \mathcal{O}(n \log n) \Rightarrow T(n) = \mathcal{O}(n \log^2 n)$

Remark. Can be improved to run in $\mathcal{O}(n \log n)$ by pre-sorting and merging lists.

3.2. **Multiplication.**

Theorem 3.3. *We can multiply two n-bit integers in* $O(n^{\log_2 3})$ *bit operations.*

Remark. This is achieved recursively, putting $x = 2^{n/2}x_1 + x_0$, $y = 2^{n/2}y_1 + y_0$ and computing $xy = 2^n x_1 y_1 + 2^{n/2}\left((x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0\right) + x_0 y_0$, so that the expensive computation is computing 3 multiplications of two order $n/2$ integers; the shifts and additions take linear time. Note that $\log_2 3 = 1.58496...$

Theorem 3.4. *Naive matrix multiplication is* $\mathcal{O}(n^3)$*. Can be improved to* $\mathcal{O}(n^{2.81})$ *by Strassen, and has been improved to* $\mathcal{O}(n^{2.376})$ *by Coppersmith-Winograd.*

4. Dynamic Programming

4.1. **Weighted interval scheduling.** The following algorithm solves the weighted interval scheduling problem. The jobs must be sorted in ascending order by finishing time. We use the recurrence

$$f(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{v_j + f(p(j)), f(j-1)\} & \text{otherwise} \end{cases}$$

where $p(j)$ is the largest index $i < j$ such that job i is compatible with job j, 0 if there is no such i.

Algorithm 4.1 Weighted interval scheduling using dynamic programming

```
1: procedure WeightedInterval(L)
2:     Sort jobs by finish times so that f_1 ≤ f_2 ≤ ··· ≤ f_n
3:     for all i = 1 to n do
4:         Compute p(i)
5:     end for
6:
7:     for all j = 1 to n do
8:         M[j] ← null
9:     end for
10:    procedure ComputeOpt(j)
11:        if M[j] is null then
12:            M[j] = max(w_j + ComputeOpt(p(j)), ComputeOpt(j − 1))
13:        end if
14:        return M[j]
15:    end procedure
16:    return ComputeOpt(n)
17: end procedure
```

Theorem 4.1. WeightedInterval *runs in* $\mathcal{O}(n \log n)$ *time.*

Remark. The runtime of WeightedInterval is $\mathcal{O}(n)$ if the jobs are already sorted by finish time and start time. We need them sorted by start time to compute the $p(j)$ in $\mathcal{O}(n)$, so we can just do one pass over the lists. Otherwise we get a $n \log n$ from doing repeated binary searches.

4.2. **Segmented least squares.** We use the recursion

$$f(j) = \begin{cases} 0 & \text{if } j = 0 \\ c + \min\limits_{1 \le i \le j} \{e(i,j) + f(i-1)\} & \text{otherwise} \end{cases}$$

where $f(j)$ is the minimum cost for points $p_1, p_2, \ldots, p_j$, and $e(i,j)$ is the minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$.

Theorem 4.2. *Computing segmented least squares runs in* $\mathcal{O}(n^3)$ *time. The bottleneck is computing* $e(i,j)$ *for all* n^2 *pairs. This can in fact be done in* $\mathcal{O}(n^2)$ *by trickery (it involves evaluating the* $e(i,j)$ *in increasing order of* $j - i$*), and brings the whole algorithm down to* $\mathcal{O}(n^2)$*.*

4.3. **Knapsack problem.**

Definition 4.3. $f(i, w)$ is the maximum profit subset of items $1, \ldots, i$ with weight limit w.

We then have the following recursion:

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

Theorem 4.4. *The dynamic programming algorithm for the knapsack problems runs in time $\mathcal{O}(nW)$ (only pseudo-polynomial).*

Remark. The decision version of knapsack is NP-complete. There is however a polynomial-time approximation algorithm that finds a feasible solution within 0.01% of the optimum.

4.4. **RNA secondary structure.**

Definition 4.5. $f(i, j)$ is the maximum number of base pairs in a secondary structure of the substring $x_i x_2 \ldots x_j$. Let $g(i, j)$ be the maximum number of base pairs in the case that x_j is aligned with some element. Then

$$g(i,j) = 1 + \max_{\substack{t: i \le t < j-4 \\ b_t, b_j \text{complementary}}} f(i, t-1) + f(t+1, j-1).$$

We use the recursion:

$$f(i,j) = \begin{cases} 0 & \text{if } i \ge j-4 \\ \max\{f(i,j-1), g(i,j)\} & \end{cases}$$

4.5. **Sequence alignment.**

Definition 4.6. $f(i, j)$ is the minimum cost of aligning strings $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$

We then have the recursion:

$$f(i,j) = \begin{cases} j\delta & \text{if } i = 0 \\ i\delta & \text{if } j = 0 \\ \min\{\alpha_{x_i,y_j} + f(i-1,j-1), & \\ \quad \delta + f(i-1,j), \delta + f(i,j-1)\} & \text{otherwise} \end{cases}$$

Theorem 4.7. *The naive implementation of the above algorithm can be implemented in $\Theta(mn)$ time and space (m and n are lengths of given strings).*

Lemma 4.8. *By calculating the recurrence iteratively (not recursively), iterating over i and only storing values of $f(i, j)$ for the current i and $i - 1$, we can find the optimal **value** in $\mathcal{O}(m + n)$ space and $\Theta(mn)$ time. It is not possible to reconstruct the alignment this way however.*

Remark. By divide and conquer, we can find the optimal alignment in linear space as well. Essentially, we determine where the shortest path in the edit distance graph intersects the middle column by iterating over the elements $(i, \lfloor \frac{n}{2} \rfloor)$ in it, summing shortest distances to the two corners $(0,0)$ and (m,n). We then recurse on the left and right subproblems.

4.6. **The Bellman-Ford algorithm for shortest paths.** Dijkstra's algorithm failed in graphs with negative edge costs. To remedy this, we introduce the Bellman-Ford algorithm.

Definition 4.9. $f(i,v)$ is the length of the shortest $v-t$ path P using at most i edges.

Then we have the recursion:

$$f(i,v) = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = t \\ \infty & \text{if } i = 0 \text{ and } v \neq t \\ \min\left\{ f(i-1,v), \min\limits_{(v,w)\in E} \{f(i-1,w) + c_{vw}\} \right\} & \text{otherwise} \end{cases}$$

By only maintaining one array $M[v]$ equal to the shortest $v-t$ path we have found so far, we can reduce the running time to $\mathcal{O}(mn)$ and space to $\mathcal{O}(m+n)$. No need to check edges of the form (v,w) unless $M[w]$ changed in the previous iteration.

4.7. **Negative cycles in a graph.**

Lemma 4.10. *If $f(n,v) = f(n-1,v)$ for all v, then there are no negative cycles.*

Lemma 4.11. *If $f(n,v) < f(n-1,v)$ for some node v, then any shortest path from v to t contains a cycle W. Moreover, W has negative cost.*

Theorem 4.12. *We can detect negative cost cycles in $\mathcal{O}(mn)$ time.*

- *Add a new node t and connect all nodes to t with a zero-cost edge.*
- *Check if $f(n,v) = f(n-1,v)$ for all nodes v.*
  - *If yes, then there are no negative cycles.*
  - *If no, then extract a cycle from shortest path from v to t.*

5. Network Flow

Definition 5.1 ($s-t$ cut)**.** An $s-t$ cut is a partition (A,B) of V with $s \in A$ and $t \in B$.

Definition 5.2. The capacity of a cut (A,B) is $\text{cap}(A,B) = \sum\limits_{e \text{ out of } A} c(e)$

Problem 5.3 (Min $s-t$ cut problem)**.** *Find an $s-t$ cut of minimum capacity.*

Definition 5.4 (Flow)**.** An $s-t$ flow is a function that satisfies:

- For each $e \in E : 0 \leq f(e) \leq c(e)$

- For each $v \in V - \{s,t\}$: $\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$

The value of a flow f is $v(f) = \sum_{e \text{ out of } s} f(e)$

Problem 5.5 (Max flow problem)**.** *Find the $s-t$ flow of maximum value.*

Lemma 5.6 (Flow value lemma)**.** *Let f be any flow, and let (A,B) be any $s-t$ cut. Then, the net flow sent across the cut is equal to the amount leaving s.*

$$\sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) = v(f)$$

Theorem 5.7 (Weak duality)**.** *Let f be any flow, and let (A,B) be any $s-t$ cut. Then the value of the flow is at most the capacity of the cut, that is $v(f) \leq cap(A,B)$.*

Corollary. *Let F be any flow, and let (A,B) be any cut. If $v(f) = cap(A,B)$, then f is a max flow and (A,B) is a min cut.*

Theorem 5.8 (Max-flow Min-cut theorem)**.** *The value of a max flow is equal to the value of the min cut.*

Definition 5.9 (Augmenting path)**. Original edge** $e = (u,v) \in E$. Flow $f(e)$, capacity $c(e)$.
Residual edge

- "Undo" flow sent.
- $e = (u,v)$ and $e^R = (v,u)$
- Residual capacity given by

$$c_f(e) = \begin{cases} c(e) - f(e) & \text{if } e \in E, \\ f(e) & \text{if } e^R \in E \end{cases}$$

Residual graph $G_f = (V, E_f)$. Consists of the residual edges with positive residual capacity.

$$E_f = \{e : f(e) < c(e)\} \cup \{e^R : c(e) > 0\}$$

Algorithm 5.1 Augments a path **P**

```
1: procedure AUGMENT(f, c, P)
2:     b ← bottleneck(P)
3:     for all e ∈ P do
4:         if e ∈ E then f(e) ← f(e) + b
5:         else f(e^R) ← f(e) − b
6:         end if
7:     end for
8:     return f
9: end procedure
```

Algorithm 5.2 Finds the maximum flow

```
1: procedure FORDFULKERSON(G, s, t, c)
2:     for all e ∈ E do f(e) ← 0
3:     end for
4:     G_f ← residual graph
5:     while there exists an augmenting path P do
6:         f ← AUGMENT(f, c, P)
7:         update G_f
8:     end while
9:     return f
10: end procedure
```

Theorem 5.10 (Running time of the above algorithm)**.** *Under the assumption that all capacities are integers between* 1 *and* C*, we have that the algorithm terminates in at most* $v(f^\star) \leq nC$ *iterations.*

Corollary. *If* $C = 1$*,* FordFulkerson *runs in* $\mathcal{O}(mn)$ *time*

Corollary. *The generic* FordFulkerson *algorithm is not polynomial in input size* $(m, n, \log C)$*. Instances can be constructed which do take* $\Theta(nC)$ *iterations.*

Theorem 5.11 (Capacity Scaling algorithm for max-flow)**.** *The following algorithm finds the max flow in* $\mathcal{O}(m \log C)$ *augmentations.*

The intuition is as follows.

- *Don't worry about finding exact highest bottleneck path.*
- *Maintain a scaling parameter* Δ*.*

- *Let $G_f(\Delta)$ be the subgraph of the residual graph consisting of only arcs with capacity at least Δ.*

Algorithm 5.3 Capacity scaling algorithm

```
1: procedure ScalingMaxFlow(G, s, t, c)
2:     for all e ∈ E do f(e) ← 0
3:     end for
4:     Δ ← smallest power of 2 greater than or equal to C
5:     G_f ← residual graph
6:     while Δ ≥ 1 do
7:         G_f(Δ) ← Δ − residual graph
8:         while there exists an augmenting path P in G_f(Δ) do
9:             f ← Augment(f, c, P)
10:            update G_f(Δ)
11:        end while
12:        Δ ← Δ/2
13:    end while
14:    return f
15: end procedure
```

Lemma 5.12. *The outer loop repeats $1+\log_2 C$ times, as Δ decreases by a factor of 2 each iteration.*

Lemma 5.13. *Let f be the flow at the end of a Δ scaling phase. Then the value of the maximum flow is at most $v(f) + m\Delta$.*

Lemma 5.14. *There are at most $2m$ augmentations per scaling phase.*

Theorem 5.15. *The* ScalingMaxFlow *algorithm finds a max flow in $\mathcal{O}(m \log C)$ augmentations. It can be implemented to run in $\mathcal{O}(m^2 \log C)$ time.*

5.1. Bipartite matching.

Problem 5.16 (Bipartite matching problem)**.** *Consider the following.*

- *Input: undirected, **bipartite** graph $G = (L \cup R, E)$.*
- *$M \subseteq E$ is a **matching** if each node appears in at most one edge in M.*
- *Max matching: find a maximum cardinality matching.*

Convert the problem to a maximum flow problem.

- Create digraph $G' = (L \cup R \cup \{s, t\}), E')$.
- Direct all edges from L to R, and assign unit capacity.

- Add sources s and unit capacity edges from each node in R to t.

Then we have the result:

Theorem 5.17. *The maximum cardinality matching in G is equal to the value of the maximum flow in G'.*

Definition 5.18 (Perfect matching)**.** A matching $M \subseteq E$ is **perfect** if each node appears in exactly one edge in M.

Theorem 5.19. *Let $G = (L \cup R, E)$ be a bipartite graph with $|L| = |R|$. Then G has a perfect matching if and only if $|N(S)| \geq |S|$ for all subsets $S \subseteq L$ ($N(S)$ is the set of nodes adjacent to nodes in S).*

5.2. **Disjoint paths.**

Problem 5.20 (Edge disjoint paths)**.** *Given a directed graph $G = (V, E)$ and two nodes s and t, find the maximum number of edge disjoint $s - t$ paths*

Definition 5.21. Two paths are **edge-disjoint** if they have no edge in common.

Theorem 5.22 (Solution)**.** *Assign unit capacity to every edge. Then the maximum number of edge disjoint $s - t$ paths equals the maximum flow value.*

Problem 5.23 (Network connectivity)**.** *Given a directed graph $G = (V, E)$ and two nodes s and t, find the minimum number of edges whose removal disconnects t from s.*

Definition 5.24. A set of edges $F \subseteq E$ disconnects t from s if all $s - t$ paths uses at least one edge in F.

Theorem 5.25. *The maximum umber of edge disjoint $s - t$ paths is equal to the minimum number of edges whose removal disconnects t from S.*

5.3. **Circulation with demands and lower bounds.**

Definition 5.26 (Circulation)**.** A **circulation** is a function that satisfies

- For each $e \in E : 0 \leq f(e) \leq c(e)$,
- For each $v \in V : \sum_{e \text{ into } v} f(e) - \sum_{e \text{ out of } v} f(e) = d(v)$.

Theorem 5.27. *Necessary condition - sum of supplies equals the sum of demands.*

$$\sum_{v \,:\, d(v) > 0} d(v) = \sum_{v \,:\, d(v) < 0} -d(v) = D$$

Theorem 5.28. *Formulate as a circulation problem.*

- *Add new sources s and sink t.*

- *For every v with $d(v) > 0$, add edge (s, v) with capacity $-d(v)$.*
- *For every v with $d(v) < 0$, add edge (v, t) with capacity $d(v)$*

Then we have the following theorem.

Theorem 5.29. *G has a circulation if and only if G' has a maximum flow over value D.*

Corollary. *Given $G = (V, E, c, d)$, there does not exist a circulation if and only if there exists a partition (A, B) of G such that $\sum_{v \in B} d(v) > cap(A, B)$.*

Theorem 5.30. *When formulating a circulation problem with lower bounds, we do the following.*

- *For every $e \in E$, end $l(e)$ units of flow along edge e.*
- *Update demands of both endpoints.*

Theorem 5.31. *There exists a circulation in G if and only if there exists a circulation in G'. If all demands, capacities and lower bounds in G are integers, then there is a circulation in G that is integer valued.*

5.4. **Survey design.**

Problem 5.32 (Survey design)**.** *We must*

- *Design survey asking n_1 customers about n_2 products.*
- *Can only survey customer i about product j if the own it.*
- *Ask customer i between c_i and c_i' questions.*
- *Ask between p_j and p_j' customers about product j.*

Design a survey that meets these specifications, if possible.

Theorem 5.33. *Formulate as a circulation problem with lower bounds.*

- *Include an edge i_j if customer owns product i.*
- *Set $L =$ set of customers, $R =$ set of products.*
- *Set $s \to i \in L$ with $f(e) = [c_i, c_i']$.*
- *Set $j \in R \to t$ with $f(e) = [p_j, p_j']$*

5.5. **Project selection.**

Problem 5.34 (Project selection)**.** *Consider the following.*

- *A set $\boldsymbol{P}$ of possible projects. Project v has revenue p_v.*
- *Set of prerequisites E. If $(v, w) \in E$, then we must do project w to do project v.*

Choose a feasible subset of projects to maximise revenue.

Definition 5.35 (Prerequisite graph)**.** A graph on the nodes of projects, with the following properties:

- Include an edge from v to w if we must do project w to do project v.
- Assign capacity ∞ to all prerequisite edges.
- Add an edge (s, v) with capacity p_v if $p_v > 0$.
- Add an edge (v, t) with capacity $-p_v$ if $p_v < 0$.
- Define $p_s = p_t = 0$.

Theorem 5.36. *(A, B) is a minimum cut if and only if $A - \{s\}$ is the optimal set of projects.*

6. Reductions

Definition 6.1 (Polynomial time reduction)**.** Problem X reduces to problem Y if arbitrary instances of problem X can be solved using:

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

We denote this as $X \leq_p Y$

Note. We can use this to classify problems according to **relative** difficulty.

Lemma 6.2. *If $X \leq_p Y$ and Y can be solved in polynomial time, then X can also be solved in polynomial time.*

Note. If $X \leq_p Y$ and $Y \leq_p X$, then we use the notation $X \equiv_p Y$. In fact the binary relation $\leq_p$ is a partial order on the set of all problems, because it is reflexive, anti-symmetric, and transitive.

There are three basic strategies for polynomial time reduction.

- Reduction by simple equivalence.
- Reduction from special case to general case.
- Reduction by encoding with gadgets.

Problem 6.3 (IndependentSet)**.** *Given a graph $G = (V, E)$ and an integer k, is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$ and for each edge, at most one of its endpoints is in S?*

Problem 6.4 (VertexCover)**.** *Given a graph $G = (V, E)$ and an integer k, is there a subset of vertices $S \subseteq V$ such that $|S| \leq k$ and for each edge, at least one of its endpoints is in S?*

Theorem 6.5. *We claim* VertexCover $\equiv_p$ IndependentSet.

Proof. S is an independent set if and only if $V - S$ is a vertex cover. □

Problem 6.6 (SetCover)**.** *Given a set U of elements, a collection $S_1, S_2, \ldots, S_m$ of subsets of U, and an integer k, does there exists a collection of at most $\leq k$ of these sets whose union is equal to U?*

Theorem 6.7. VertexCover $\leq_p$ SetCover

Proof. Given a VertexCover instance $G = (V, E), k$, we can construct a set cover instance whose size equals the size of the vertex cover instance. Set $k = k$, $U = E, S_v = \{e \in E : e \text{ incident to } v\}$. Then there is a set cover of size $\leq k$ if and only if there is a vertex cover of size $\leq k$. □

Theorem 6.8. 3-sat $\leq_p$ IndependentSet

Proof. Given an instance Φ of 3-sat, we construct an instance (G, k) of IndependentSet that has an independent set of size k if and only if Φ is satisfiable.

- In each clause, connect all literals in a triangle.
- Connect literal to each of its negations.

□

Theorem 6.9 (Self-reducibility)**.** *Consider the following.*

- ***Decision problem.*** *Does there exist a vertex cover of size $\leq k$?*
- ***Search problem.*** *Find vertex cover of minimum cardinality.*

*By self-reducibility, we have that the search problem reduces to the decision problem. That is, search problem $\leq_p$ decision problem. This applies to all (**NP**-complete) problems in this chapter. This is dependent on the number of possible values of the optimal (you cannot binary search an infinite set).*

7. Complexity

Definition 7.1 (Complexity class **P**)**.** The class of decision problems for which there is a polynomial time algorithm.

Definition 7.2 (Certifier)**.** An algorithm $C(s, t)$ is a certifier for a problem X if for every string s, $s \in X$ if and only if there exists a string t such that $C(s, t)$ outputs true. The string t is known as a *certificate* or *witness*.

Definition 7.3 (Complexity class **NP**)**.** The class of decision problems for which there is a polynomial time certifier - an algorithm $C(s, t)$ that is polynomial in time and $|t| \leq p(|s|)$ for some polynomial **P**.

Remark. **NP** stands for **nondeterministic** polynomial time.

Example 7.4 (Certifiers and Certificates - sat)**.** Consider the problem sat - given a CNF (conjunctive normal form - the '*and* combination of many *or* clauses') formula Φ, is there a satisfying assignment?

- **Certificate.** An assignment of truth values to the n boolean variables.
- **Certifier.** Check that each clause in Φ has at least one true literal.

Conclusion: sat is in **NP**.

Example 7.5 (Certifiers and Certificates - HamiltonianCycle)**.** Consider the problem HamiltonianCycle - given an undirected graph $G = (V, E)$, does there exist a simple cycle C that visits every node?

- **Certificate.** An permutation of the n nodes.
- **Certifier.** Check that the permutation contains each node in V exactly once, and that there is an edge between each pair of adjacent nodes in the permutation.

Conclusion: HamiltonianCycle is in **NP**.

Definition 7.6 (Complexity class **EXP**)**.** The class of decision problems for which there is an exponential time algorithm

Theorem 7.7.

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$$

7.1. **NP-Completeness.**

Definition 7.8 (Polynomial transformation)**.** A problem X **polynomial transforms** to problem Y if given any input x to X, we can construct an input y in polynomial time such that x is a TRUE instance of X if and only if y is a TRUE instance of Y. $|y|$ must be polynomial in $|x|$ as it is constructed in polynomial time. The definition of transformation is due to Karp, reduction to Cook.

Remark. A polynomial transformation is a polynomial reduction with just one call to oracle for Y, exactly at the end of the algorithm for X. Almost all previous reductions were of this form.

Definition 7.9 (**NP**-complete)**.** A problem Y in **NP** is **NP**-complete if for every problem X in **NP**, $X \leq_p Y$.

Theorem 7.10. *Suppose Y is **NP**-complete. Then Y is solvable in polynomial time if and only if **P** = **NP**.*

Example 7.11 (Establishing a problem Y is **NP**-complete)**.** The following is sufficient.

- Show that Y is in **NP**.
- Choose an **NP**-complete problem X.
- Show that $X \leq_p Y$.

Theorem 7.12. 3-SAT *is **NP**-complete*

Proof. CIRCUIT-SAT $\leq_p$ 3-SAT □

Corollary. VertexCover*,* IndependentSet*, and* SetCover *are all **NP**-complete.*

Proof. By reductions in the first part, we had that 3-SAT reduces to all of the above problems. Hence, they are all **NP**-complete. □

7.2. **co-NP and the Asymmetry of NP.**

Definition 7.13. Given a decision problem X, its complement $\overline{X}$ is the same problem with the TRUE and FALSE answers reversed.

Definition 7.14 (Complexity class **co-NP**)**.** Complements of decision problems in **NP**.

Question 7.15. *Does* $\boldsymbol{NP = co\text{-}NP}$*?*

Theorem 7.16. *If* $\boldsymbol{NP \neq co\text{-}NP}$*, then* $\boldsymbol{P \neq NP}$*.*

Theorem 7.17. $\boldsymbol{P \subseteq NP \cap co\text{-}NP}$

Theorem 7.18. PRIMES *is in* $\boldsymbol{NP \cap co\text{-}NP}$*. In fact* PRIMES *is in* $\boldsymbol{P}$ *(AKS 2002).*

Theorem 7.19 (FACTOR is in **NP** $\cap$ **co-NP**)**.** *Consider the following problems.*
FACTORIZE. *Given an integer x, find its prime factorisation.*
FACTOR. *Given two integers x and y, does x have a nontrivial factor less than y?*

Theorem 7.20. FACTOR $\equiv_p$ FACTORIZE

Then we have FACTOR *is in* $\boldsymbol{NP \cap co\text{-}NP}$

We have established the following sequence.

$$\text{PRIMES} \leq_p \text{COMPOSITES} \leq_p \text{FACTOR}$$

Question 7.21. *Does* FACTOR $\leq_p$ PRIMES*?*

Definition 7.22 (**NP**-hard)**.** A decision problem such that every problem in **NP** reduces to it. This problem is not necessarily in **NP**.

8. Dealing with Intractability

Here, we attempt to solve special cases of **NP**-complete problems that arise in practice.

8.1. **Vertex Cover.**

Theorem 8.1. *The following algorithm determines if G has a vertex cover of size less than or equal to k in $\mathcal{O}(2^k kn)$ time.*

Algorithm 8.1 Small vertex covers

```
1: procedure VertexCover(G, k)
2:     if G contains no edges then return True
3:     end if
4:     if G contains more than k|G| edges then return False
5:     end if
6:
7:     Let (u, v) be any edge of G
8:     a = VertexCover(G − {u}, k − 1)
9:     a = VertexCover(G − {v}, k − 1)
10:    return a or b
11: end procedure
```

Theorem 8.2. *The above algorithm runs in time $\mathcal{O}(2^k kn)$.*

8.2. **Independent set on trees.**

Theorem 8.3. *The following greedy algorithm finds a maximum cardinality independent set in forests (and hence trees).*

Algorithm 8.2 Finds an independent set in a tree or forest.

```
1: procedure IndependentSet(F)
2:     S ← φ
3:     while F has at least one edge do
4:         Let e = (u, v) be an edge with v a leaf.
5:         S ← S ∪ {v}
6:         F ← F − {u, v}
7:     end while
8:     return S
9: end procedure
```

Theorem 8.4. *The above algorithm can run in time $\mathcal{O}(n)$ by considering nodes in postorder.*

8.3. **Weighted independent set on trees.**

Problem 8.5. *Given a tree and node weights $w_v > 0$, find an independent set S that maximises $\sum_{v \in S} w_v$.*

Theorem 8.6. *Let $f_{in}(u)$ be the maximum weight independent set rooted at u containing u, and let $f_{out}(u)$ be the maximum weight independent set rooted at u not containing u. Then we have the following:*

$$f_{in}(u) = w_u + \sum_{v \in \text{CHILD}(u)} f_{out}(v)$$
$$f_{out}(u) = \sum_{v \in \text{CHILD}(u)} \max\{f_{in}(v), f_{out}(v)\}$$

We can find the maximum weighted set by rooting the tree at a node r and considering each node in postorder, and calculating f_{in} and f_{out} at each node v. This takes $\mathcal{O}(n)$ time.

9. Randomized and approximation algorithms

9.1. **Probability results.** The union bound is $P(\cup_i X_i) \leq \sum_i P(x_i)$.
The Markov inequality for a non-negative X is $P(X \geq a) \leq \frac{\mathbb{E}[X]}{a}$.
Chebyshev's inequality is $P(|X - \mathbb{E}[X]| \geq a) \leq \frac{\text{Var}(X)}{a^2}$.
The horrific Chernoff bound is $\sum_{i=\lfloor \frac{n}{2} \rfloor + 1}^{n} \binom{n}{i} p^i (1-p)^{n-i} \geq 1 - e^{-2n\left(p - \frac{1}{2}\right)^2}$.

9.2. **Randomized algorithm for global minimum cut.** The following algorithm, known as the Contraction algorithm, finds the global minimum cut in an undirected graph.

Definition 9.1 (Global minimum cut)**.** The size of a cut (A, B) of a graph G is the number of edges with one end in A and one end in B.

Algorithm 9.1 The Contraction algorithm for finding the global minimum cut

```
1: procedure CONTRACTION(G)
2:     for all v ∈ V do S[v] ← v
3:     end for
4:     if |G| = 2 then return the cut (S[v1], S[v2])
5:     else
6:         choose an edge e = (u, v) from G uniformly at random.
7:         Set G' equal to the graph resulting from the contraction of e, with a new node w replacing
       u and v.
8:         S[w] ← S[u] ∪ S[v]
9:         return CONTRACTION(G')
10:    end if
11: end procedure
```

Theorem 9.2. *The algorithm returns the global minimum cut with probability* $\binom{n}{2}^{-1}$. *Running the algorithm* $\binom{n}{2}\log n$ *returns a global minimum cut with probability greater than* $1 - \frac{1}{n}$

9.3. **Approximation algorithm for makespan scheduling.**

Problem 9.3 (Makespan scheduling)**.** *We have n jobs, each of which takes time* t_i *to process, and m machines. Let* A_j *be the set of jobs assigned to machine j. Let* $L_j = \sum_{i \in A_j} t_i$ *be the* ***load*** *of machine j. The* makespan *of an assignment is the maximum load on any machine.*

Algorithm 9.2 Longest Processing Time (LPT) makespan approximation

```
1: procedure LPT(J)
2:     Sort jobs J so that t_1 ≥ t_2 ≥ ··· ≥ t_n.
3:     A_j ← ∅
4:     L_j ← 0.
5:     for i = 1 to n do
6:         j ← argmin_k L_k
7:         A_j ← A_j ∪ i
8:         L_j ← L_j + t_i
9:     end for
10: end procedure
```

Theorem 9.4. *If there are at most m jobs,* LPT *scheduling is optimal.*

Theorem 9.5. LPT *scheduling is a* $\frac{3}{2}$*-approximation. Careful analysis shows that* LPT *scheduling is a* $\frac{4}{3}$*-approximation*

9.4. **Randomized algorithm for max-3-sat.**

Problem 9.6 (MAX-3-SAT)**.** *Given a* 3-SAT *instance, find a truth assignment that satisfies as many clauses as possible.*

Lemma 9.7. *As* 3-SAT *is* ***NP****-complete, this is an* ***NP****-hard search problem.*

Theorem 9.8. *We can find an approximate solution by setting each variable* TRUE *with probability* $\frac{1}{2}$. *The expected number of clauses satisfied by a random assignment is within a factor of* $\frac{7}{8}$ *of the optimal solution.*

Proof. Each clause is satisfied with probability $1 - \left(\frac{1}{2}\right)^3 = \frac{7}{8}$. Let Z_i be a random variable equal to 1 if clause i is satisfied, and 0 otherwise. Then, by linearity of expectation, we have

$$\mathbb{E}[Z] = \mathbb{E}[Z_1] + \mathbb{E}[Z_2] + \cdots + \mathbb{E}[Z_k] = \frac{7}{8}k$$

Since no assignment can satisfy more than k clauses. □

9.5. **Randomized algorithm for database access.**

Problem 9.9. *Suppose we have n processes $P_1, \ldots, P_n$ attempting to access a single shared database. The database has the property that it can be accessed by at most one process in a single time period, and if two or more processes attempt to access the database, all processes are locked out for the period.*

Theorem 9.10. *Using a randomized algorithm where a process attempts to access the database with probability $\frac{1}{n}$ at each time step, we have the following: with probability at least $1 - \frac{1}{n}$, all processes succeed in accessing the database at least once within $t = e\lceil en \rceil \ln n$ rounds.*

9.6. **Approximation algorithm for the travelling salesman problem.** In the case of the metric-TSP, where the edge weights satisfy the triangle inequality, we have the following simple algorithm for a 2-approximation for TSP.

Algorithm 9.3 2-Approximation for METRIC-TSP

```
1: procedure APPROX-METRIC-TSP(G)
2:     Compute a minimal spanning tree T of G
3:     Root T arbitraryily and traverse in pre-order (i.e. DFS): v_1, v_2, ..., v_n.
4:     return TOUR: v_1 → v_2 → ··· → v_n → v_1
5: end procedure
```

Lemma 9.11. *Let $\mathrm{A}(G)$ be the approximation returned by the above algorithm on the graph G. Then $\mathrm{A}(G) \leq 2 \times \mathrm{MST}(G)$.*

Lemma 9.12. *Let $\mathrm{OPT}(G)$ be the optimal solution for our* METRIC-TSP *instance. Then $\mathrm{MST}(G) \leq \mathrm{OPT}(G)$.*

Theorem 9.13. *We then have $\mathrm{A}(G) \leq 2 \times \mathrm{OPT}(G)$ - that is, the above algorithm is a 2-approximation for* METRIC-TSP

9.7. **Some NP-complete problems.** The following are some decision problems which are NP-complete.

- 3SAT
- Vertex cover
- Independent set
- Clique
- Travelling salesman
- Hamiltonian path
- Hamiltonian circuit
- Max cut
- Longest path
- Knapsack
- Subset sum