Algorithms II Cheat-Sheet

Notation

$A \in [10] \equiv A \in [1..10]$
{a, b c} is a set of vertices
G{a, b, c} is a graph
$\overrightarrow{x}$ just represents a vector x

Big O

Notation	Intuitive meaning	Analogue
$f(n) \in O(g(n))$	f grows at most as fast as g	$\leq$
$f(n) \in \Omega(g(n))$	f grows at least as fast as g	$\geq$
$f(n) \in \Theta(g(n))$	f at the same rate as g	$=$
$f(n) \in o(g(n))$	f grows strictly less fast than g	$<$
$f(n) \in \omega(g(n))$	f grows strictly faster than g	$>$

Notation	Formal definition
$f(n) \in O(g(n))$	$\exists C,\ n_0 \colon \forall n \geq n_0 \colon f(n) \leq C \cdot g(n)$
$f(n) \in \Omega(g(n))$	$\exists c,\ n_0 \colon \forall n \geq n_0 \colon f(n) \geq c \cdot g(n)$
$f(n) \in \Theta(g(n))$	$\exists c,\ C,\ n_0 \colon \forall n \geq n_0 \colon c \cdot g(n) \leq f(n) \leq C \cdot g(n)$
$f(n) \in o(g(n))$	$\forall C \colon \exists n_0 \colon \forall n \geq n_0 \colon f(n) \leq C \cdot g(n)$
$f(n) \in \omega(g(n))$	$\forall c \colon \exists n_0 \colon \forall n \geq n_0 \colon f(n) \geq c \cdot g(n)$

Interval Scheduling

A **request** is a pair of integers (s, f) with $0 \leq s \leq f$.
We call s the **start time** and f the **finish time**.

A set A of requests is **compatible** if for all distinct $(s, f), (s', f') \in A$, either $s' \geq f$ or $s \geq f'$ — that is, the requests' time intervals don't overlap.

Interval Scheduling Problem
Input: An array $\mathcal{R}$ of n requests $(s_1, f_1), \ldots, (s_n, f_n)$.
Desired Output: A compatible subset of $\mathcal{R}$ of maximum possible size.

Algorithm: GreedySchedule
Input: An array $\mathcal{R}$ of n requests.
Output: A maximum compatible subset of $\mathcal{R}$.

```
1 begin
2   Sort R's entries so that R ← [(s_1, f_1), ..., (s_n, f_n)] where f_1 ≤ ··· ≤ f_n.
3   Initialise A ← [], lastf ← 0.
4   foreach i ∈ {1, ..., n} do
5     if s_i ≥ lastf then
6       Append (s_i, f_i) to A and update lastf ← f_i.
7   Return A.
```

Complexity:
Step 2 takes O(n log n)
Steps 3–6 all take O(1) time and are executed at most n times.
$\therefore\ totalrunningtime = O(nlogn) + O(n)O(1) = O(nlogn)$.

Interval Scheduling

Formal GreedySchedule:
$A^+ := \text{argmin}\ \{f : (s, f) \in R, A \cup \{(s, f)\}$ is compatible} for all $A \subseteq R$,
$A_0 := \emptyset, \qquad A_{i+1} := A_i \cup \{A_i^+\}$
t := max{i: A_i is defined}

Interval Scheduling Proofs

Lemma: Greedy Schedule always outputs A_t
Proof: By induction form the following loop invariant. At the start of the i'th iteration of 4-7:

- A is equal to $A_t \cap \{(s_1, f1), ..., (s_{i-1}, f_{i-1})\}$
- lastf is equal to the latest finish time of any request in A (or 0 if A = [])

Lemma: A_t is a compatible set
Proof: Instant by induction; A_0 is compatible, and if A_i is compatible then so is $A_{i+1} = A_i \cup A^+$ by the definition of A_i^+

Lemma: A_t is a maximum compatible subset of the Array R (look in pseudocode)
Proof:
Base case for i = 1: A_0^+ is the fastest finishing request in R by definition
Inductive step: Suppose A_i finishes faster than B_i.
Let B_i^+ be the (i+1)'st fastetst-finishing element of B. Since A_i finishes faster than B_i, $A_i \cup \{B_i^+\}$ is compatible. Hence by definition, A_i^+ exists and finishes no later than B_i^+

Theorem: GreedySchedule outputs A_t, which is a maximum compatible set.
Proof: putting all of the above proofs together, we prove the theorem.

Graph Theory

Graph: G = (V, E)
Edge: E = E(G) is a set of edges contained in $\{\{u, v\} : u, v \in V, u \neq v\}$
Vertex: V = V(G) is a set of vertices
Subgraph: H = (V_H, E_H) of G is a graph with $V_H \subseteq V$ and $E_H \subseteq E$
Induced Subgraph: is a subgraph if $E_H = \{e \in E : e \subseteq V_H\}$
Component: H of G is a maximal connected induced subgraph of G.
Degree: $d(v) = |N(v)|$
Neighbourhood: $N(v) = \{w \in V : \{v, w\} \in E\}$
Walk: sequence of vertices $v_0 ... v_k$ such that $\{v_i, v_{i+1}\} \in E$ for all i $\leq$ k-1
Length: the value of k (see above walk definition)
Euler Walk: a walk that contains every edge in G exactly once.
Isomorphism: two graphs are isomorphic if there is a bijection f: $V_1 \to V_2$ such that $\{f(u), f(v)\} \in E_2$ if and only if $\{u, v\} \in E_1$
Path: is a walk in which no vertices repeat
Connected: A graph is connected if any two vertices are joined by a path
Digraph: is a pair G = (V, E), V is a set of vertices and E is a set of edges contained in $\{(u, v) : u, v \in V, u \neq v\}$
Strongly connected: G is .. if for all $u, v \in V$, there is a path from u to v and a path from v to u.
Weakly connected:
In-Neighbourhood: $N^-(v) = \{u \in V(G) : (u, v) \in E(G)\}$
Out-Neighbourhood: $N^+(v) = \{w \in V(G) : (v, w) \in E(G)\}$
Cycle: is a walk $W = w_0 ... w_k$ with $w_0 = w_k$ and $k \geq 3$, in which every vertex appears at most once except for w_0 and w_k (which appear twice)
Hamilton cycle: is a cycle containing every vertex in the graph
k-regular: a graph is .. if every vertex has degree k
Bijection:
Planar: a graph is planar if it can be drawn without any two edges overlapping.

Graph Theory

Theorem: If G has an Euler walk, then either:

- every vertex of G has even degree; or
- all but two vertices v_0 and v_k have even degree, and any euler walk must have v_0 and v_k as endpoints

---

Theorem: let G = (V, E) be a digraph with no isolated vertices, and let $U, v \in V$. Then G has an Euler walk from u to v if and only if G is weakly connected and either:

- u = v and every vertex of G has equal in- and out-degrees; order
- $u \neq v, d^+(u) = d^-(u) + 1, d^-(v) = d^+(v) + 1$ and every other vertex of G has equal in- and out-degrees

---

Dirac's Theorem: Let $n \geq 3$. Then any n-vertex graph G with minimum degree at least $\frac{n}{2}$ has a Hamilton cycle.

Handshake lemma: For any graph G = (V, E), $\sum_{v \in V} d(v) = 2|E|$

Proof: All edges contain two vertices, and eachvertex v is in d(v) edges. Count the number of verted-edge pairs: Let $X = \{(v, e) \in V \times E : v \in E\}$. Then $|X| = 2|E|$ and $|X| = \sum_{v \in V} d(v)$, so we're done.

Directed Handshake lemma: For any graph G = (V, E), $\sum_{v \in V} d^+(v) = \sum_{v \in V} d^-(v) = 2|E|$

Proof: TODO. Instead of counting vertex-edge pairs, we count tail-edge pairs. Each edge has one tail so $|X| = |E|$

Trees

Forest: a graph with no cycles

Tree: a forest that is connected

Root: for T = (V, E). Root $r \in V$ as follows. For all vertices $v \neq r$, let P_v be the unique path from r to v. Then direct each P_v from r to v.

Leaf: is a degree-1 vertex. Root cannot be a leaf.

Ancestor: u is an .. of v if u i on P_v

Parent: u is the .. of v if $u \in N^-(v)$

level: first .. L_0 of T is r, and $L_{i+1} = N^+(L_i)$.

depth: of T is max{i: $L_i \neq \emptyset$}. Root doesn't count

---

Lemma 1: If T = (V, E) is a tree, then any pair of vertices u, v ∈ V is joined by a unique path uTv in T.

Lemma 2: Any n-vertex tree has n-1 edges

Lemma 3: Any n-vertex tree T = (V, E) with $n \geq 2$ has at least 2 leaves

---

Tree Properties:

A: T is connected and has no cycles

B: T has n-1 edges and is connected

C: T has n-1 edges and has no cycles

D: T has a unique path between any pair of vertices

$A \implies B, C, D$ $\qquad A \impliedby B, C, D.$

Depth First Search

Graphs as data structures:

Adjacency Matrix:

Storing: $\Theta(|V|^2)$ space

Adjacency query: $\Theta(1)$ time

Neighbourhood query: $\Theta(|V|)$ time

Adjacency List:

$\boxed{s} \rightarrow b, a$

$\boxed{a} \rightarrow s, c$

Storing: $\Theta(|V| + |E|)$ space

Adjacency query: $\Theta(d^+(u))$ time

Neighbourhood query: $\Theta(d^+(u))$ time

DFS:

Input : Graph $G = (V, E)$, vertex $v \in V$.

Output : List of vertices in v's component.

```
1  Number the vertices of G as v_1, ..., v_n.
2  Let explored[i] ← 0 for all i ∈ [n].
3  Procedure helper(v_i)
4      if explored[i] = 0 then
5          Set explored[i] ← 1.
6          for v_j adjacent to v_i do
7              if explored[j] = 0 then
8                  Call helper(v_j).
9  Call helper(v).
10 Return [v_i : explored[i] = 1] (in some order).
```

Complexity: In total there are $\sum_{v \in V} d(v) = O(|E|)$ calls to helper (each vertex only runs lines 5-7 once), and there is O(1) time between calls. So the running time is $O(|V| + |E|)$.

Invariant: When helper is called, if explored[i] = 1 then $v_i \in V(C)$.

Claim: Every vertex in P is explored

Proof by induction: We prove $x_1, ..., x_i$ are explored for all $i \leq t$. x_1 is explored. If x_i is explored, then helper(x_{i+1}) will be called from helper(x_i). so x_{i+1} will also be explored.

DFS Tree: a .. T of G is a rooted tree satisfying:

- $V(T)$ is the vertex set of a component of G;
- If $\{x, y\} \in E(G)$, then x is an ancestor of y in T or vice versa.

Breadth First Search

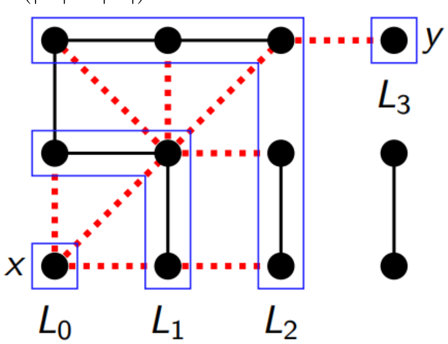
Distance: The distance between x and y, d(x, y), is the length in edges of a shortest path between x and y, or ∞ if no such path exists.

BFS:

Input : Graph $G = (V, E)$, vertex $v \in V$.
Output : $d(v, y)$ for all $y \in V$ and "a way of finding shortest paths".

```
Number the vertices of G as v = v_1, ..., v_n.
Let L[i] ← ∞ for all i ∈ [n].
Let L[1] ← 0, pred[1] ← None.
Let queue be a queue containing all tuples
  (v, v_j) with {v, v_j} ∈ E.
while queue is not empty do
    Remove front tuple (v_i, v_j) from queue.
    if L[j] = ∞ then
        Add (v_j, v_k) to queue for all
          {v_j, v_k} ∈ E, k ≠ i.
        Set L[j] ← L[i] + 1, pred[j] = i.
Return L and pred.
```

Complexity: If G is in adjacency list form, each edge is added to queue at most twice, incurring $O(1)$ overhead each time, so the running time is $O(|V| + |E|)$.



BFS explanation: BFS works by starting at a vertex and then adding all adjacent vertices to a queue. We then take the first vertex in the queue and look for a new set of adjacent vertices to add, repeating the process until we have reached our destination.

Dijkstra's Algorithm

Weighted Graph: is a pair (G, w), where G is a graph and $w : E(G) \to \mathbb{R}$ is a **weight function**

Length: of a path/walk $P = x_1 \dots x_t$ is the total weight of P's edges: length(P) $= \sum_{i=1}^{t-1} w(x_i, x_{i+1})$.

Distance: from x to y is the shortest length of any path/walk from x to y, or ∞ if they are in different components

Priority queue: each element has a priority, and the first element is the one with the lowest priority.

Dijkstra:

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

```
Number the vertices of G as v = v_1, ..., v_n.
queue ← StartQueue(n).
foreach i = 1 to n do
    dist[i] ← ∞ and call queue.Insert(v_i, ∞).
Call queue.ChangeKey(v_1, 0).
do
    vert ← queue.Extract(), say vert = v_i.
    foreach (v_i, v_j) ∈ E do
        dist[j] ← min{dist[j], dist[i] + w(i, j)}.
        Call queue.ChangeKey(v_j, dist[j]),
while queue is not empty
Return dist.
```

Complexity: We perform $O(|V|)$ Insert operations and Extract operations, and $O(|E|)$ ChangeKey operations, for a total of $O((|V| + |E|)log|V|)$ time when G is given in adjacency list form.

Dijkstra Operations:

- StartQueue(n) returns a new priority queue of maximum length n.
- Insert(x, p) inserts a new element x with priority p.
- Exctract() removes and returns the lowest-priority element.
- ChangeKey(x, p) udpates the priority of x to p.
- StartQueue takes $O(n)$ time, all other operations take $O(log(n))$ time.

Matchings

Matching: A matching in a graph is a collection of disjoint edges. It is **perfect** if every vertex is contained in some matching edge.

Lemma: G is bipartite if and only if it has no odd-length cycle.

Proof of "only if": Suppose G has bipartition (A, B) and $C = v_1 \dots v_k$ is an odd cycle in G.

Wlog $v_1 \in A$. Since A has no edges, this means $v_2 \in B$. Continuing the argument, $v_i \in A$ if i is odd, and $v_i \in B$ if i is even. In particular, $v_k \in A$.

But v_1 and v_k are adjacent, so this is a contradiction. ✓

Bipartite Graph: A graph G = (V , E) is bipartite if V can be partitioned into disjoint sets A and B which contain no edges

MaxMatching:

```
1 begin
2     Find a bipartition (A, B) of G. Initialise M ← [].
3     repeat
4         Form the graph D_{G,M}.
5         Set P to be a path from U ∩ A to U ∩ B in D_{G,M} if one exists.
            Otherwise, break.
6         Update M ← Switch(M, P).
7     Return M.
```

Complexity:

- Steps 2, 4 and 6 can all be done in $O(|E|)$ time. (Exercise!)
- Step 5 can be done in $O(|E|)$ time using breadth-first search, if G is in adjacency-list form.
- Steps 4–6 repeat at most $|V|$ times.

So overall the running time is $O(|E||V|)$.

Berg's Lemma: M has no augmenting paths $\implies$ M is maximum.

Prim's Algorithm

Prim's Algorithm explanation: We work greedily: pick an arbitrary start vertex, then grow it into a spanning tree by always choosing one of the cheapest available edges.

Formal explanation:

Formally: Let $T_1 = (\{v\}, \emptyset)$ for some arbitrary $v \in V$.

Let E_i be the set of edges from $V(T_i)$ to $V \setminus V(T_i)$.

Form T_{i+1} by adding a lowest-weight edge $e_i \in E_i$ to T_i, so

$$V(T_{i+1}) = V(T_i) \cup e_i \text{ and } E(T_{i+1}) = E(T_i) \cup \{e_i\}.$$

Prim's algorithm is to calculate and return $T_{|V|}$. Why does this work?

Prim pseudocode:

Input : **Connected weighted** graph $G = ((V, E), w)$.
Output : **A minimum spanning tree for G**.

Number the vertices of G **arbitrarily** as $v_1, \ldots, v_n$.
Let $\mathtt{L}[i] \leftarrow \infty$ for all $i \in [n]$.
Let $\mathtt{L}[1] \leftarrow 0$, $\mathtt{pred}[1] \leftarrow \mathtt{None}$.
Let queue be a **length-$|E|$ priority** queue containing all tuples (v, v_j) with $\{v, v_j\} \in E$,
  using their edge weights as priorities.
while queue *is not empty* **do**
  Remove front tuple (v_i, v_j) from queue.
  if $\mathtt{L}[j] = \infty$ **then**
    Add (v_j, v_k) to queue for all $\{v_j, v_k\} \in E$, $k \neq i$.
    Set $\mathtt{L}[j] \leftarrow \mathtt{L}[i] + 1$, $\mathtt{pred}[j] = i$.
Return pred.

Complexity:

Time analysis: As with breadth-first search, each edge is only processed twice. Processing each edge now takes $\Theta(\log|E|)$ worst-case time, so overall the algorithm runs in $O(|E|\log|E|)$ time. (Note $|E| \geq |V|$.)

Kruskal's Algorithm

Kruskal's Algorithm explanation: Rather than picking the lowest-weight edge that grows our component, we just pick the lowest-weight edge anywhere that doesn't make a cycle.

Formal explanation:

Formally: Let $e_1, \ldots, e_m$ be the edges of G, with $w(e_1) \leq \cdots \leq w(e_m)$.

Let $T_0 = (V, \emptyset)$ be the empty graph on V.
Given T_i, let $T_{i+1} = T_i + e_{i+1}$ if this is a forest, or T_i otherwise.

Kruskal's algorithm is to calculate and return T_m. Why does this work?

Kruskal pseudocode:

Input : Connected weighted graph $G = ((V, E), w)$ in adjacency list form.
Output : A minimum spanning tree for G.

1 Sort the edges by weight as $e_1, \ldots, e_m$, with $w(e_1) \leq \cdots \leq w(e_m)$.
2 Let $T \leftarrow (V, \emptyset)$ be the empty tree on V.
3 Let $\mathcal{C} = \mathtt{MakeUnionFind}(V)$.
4 **for** $i = 1$ *to* m **do**
5   Write $e_i \to \{u_i, v_i\}$.
6   **if** $\mathcal{C}.\mathtt{FindSet}(u_i) \neq \mathcal{C}.\mathtt{FindSet}(v_i)$ **then**
7   Let $T \leftarrow T + e_i$.
8   Call $\mathcal{C}.\mathtt{Union}(u_i, v_i)$.
9 Return T.

Complexity:

Now line 3 takes $O(|V|)$ time, and each iteration of lines 6 and 8 takes $O(\log|V|)$ time.

So overall, since G is connected and $|E| \geq |V| - 1$, the running time is $O(|E|\log|V|)$ — exactly what we got from Prim's algorithm!

Union-find data structure

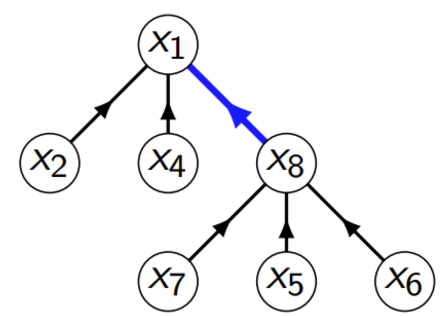
Union-find data structure Operations:
MakeUnionFind(X): Makes a new union-find data structure containing a 1-element set x for each element $x \in X$. Takes O(—X—) time.

Union(x, y): Merge the set containing x with the set containing y into a single set in the data structure. Takes O(log —X—) time.

FindSet(x): Returns a unique identifier for the set containing x. Takes O(log —X—) time.

Union/2-3-4 tree:

$\mathtt{Union}(x_4, x_7);$



Insertion:

To insert a value k, first we find the leaf that would contain it if it was there. If it's a 2-node or a 3-node, we can just add the new value.

If it's a 4-node, we first **split** it, sending one value up to its parent and keeping the others as 2-nodes.

If its parent is a 4-node as well, we're in trouble... so we split all 4-nodes we find on the way down. Still only takes $O(d)$ time.

If we have to split the root, d increases by 1. But balance is maintained!

Searching:

Say a 3-node has values $x_1 \leq x_2$, and children c_1, c_2 and c_3.

Then all descendants of c_1 must have values at most x_1...
All descendants of c_2 must have values greater than x_1 and less than x_2...
And all descendants of c_3 must have values greater than x_3.

4-nodes work the same way. So we can still find a value in $O(d)$ time.

Deletion:

If v is a 2-node with a 3-node or 4-node sibling w, we **transfer** a value from w to v, reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w, and a 3-node or 4-node parent, we **fuse** v, w and a value from v's parent, reducing to the 4-node case.

non-leaf-deletion:

Exercise: If v is not stored in a leaf, then the **predecessor** w of v — the value just before v in sorted order — will always be in a leaf.

So we can overwrite v with w, and then delete w from its leaf — leaving the structure of the tree untouched!

Linear Programming

Feasible: We say $\vec{x} \in \mathbb{R}^n$ is a feasible solution if $\vec{x} \geq \vec{0}$ and $A\vec{x} \leq \vec{b}$.
Optimal: We say $\vec{x}$ is an optimal solution if $f(\vec{y}) \leq f(\vec{x})$ for all feasible $y \in \mathbb{R}^n$
Polytope: is a geometric object with flat sides
Corollary: There will always be an optimal solution

Non-Standard Form:
$-4x + 5y - z \to$ max subject to
$x + y + z \leq 5;$
$x + y + z \geq 5;$
$x + 2y \geq 2;$
$x, z \geq 0.$

Standard Form:
$-4x + 5(y_1 - y_2) - z \to$ max subject to
$x + (y_1 - y_2) + z \leq 5;$
$-x - (y_1 - y_2) - z \leq -5;$
$-x - 2(y_1 - y_2) \leq -2;$
$x, y_1, y_2, z \geq 0.$

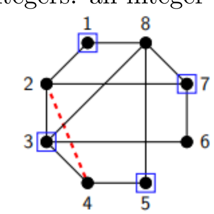
Matrix Form:

$$-4x + 5y_1 - 5y_2 - z \to \text{max subject to}$$

$$\begin{pmatrix} 1 & 1 & -1 & 1 \\ -1 & -1 & 1 & -1 \\ -1 & -2 & 2 & 0 \end{pmatrix} \begin{pmatrix} x \\ y_1 \\ y_2 \\ z \end{pmatrix} \leq \begin{pmatrix} 5 \\ -5 \\ -2 \end{pmatrix};$$

$$x,\ y_1,\ y_2,\ x \geq 0.$$

Simplex Method: Search greedily for a vertex of the feasible polytope which maximises the objective function
Worst case: hypercube which has $\Omega(2^n)$ vertices.
In practice it only need $\Theta(n)$ steps
Vertex Cover: in a graph G, is a set $X \subseteq V$ such that every edge in E has at least one vertex in X
We can express finding a minimum vertex cover as solving a linear program in which the solutions must be integers: an integer linear program.



$\sum_v x_v \to$ min subject to
$x_u + x_v \geq 1$ for all $\{u, v\} \in E$;
$x_v \leq 1$ for all $v \in V$;
$x_v \geq 0$ for all $v \in V$;
$x_v \in \mathbb{N}$ for all $v \in V$.

$X = \{1, 3, 5, 7\}$ is **not** a vertex cover.

Here we have $x_1 = x_3 = x_5 = x_7 = 1$ and $x_0 = x_2 = x_4 = x_6 = 0$.

The uncovered edge $\{2, 4\}$ corresponds to the constraint $x_2 + x_4 \geq 1$, which is violated.

Flow Networks

Flow Network: consists of a directed graph $G = (V, E)$, a **capacity function** $c : E \to \mathbb{N}$, a **source** vertex $s \in V$ with $N^-(s) = \emptyset$, and a **sink** vertex $t \in V$ with $N^+(t) = \emptyset$

Flow: is a function in (G, c, s, t) $f : E \to \mathbb{R}$ with properties:

- No edge has more flow than capacity; formally, for all $e \in E, 0 \leq f(e) \leq c(e)$
- Flow is conserved at vertices; flow in = flow out

Maximum Flow: a flow f maximising the value of the flow, $v(f)$

Cut: is any pair of disjoint edges $A, B \subseteq V$ with $A \cup B = V$, $s \in A$ and $t \in B$.

Lemma 1: For all sets $X \subseteq V\ \{s,t\}$, we have $f^+(X) = f^-(X)$. So flow is conserved in sets/cuts as well as vertices

Proof: By summing conservation of flow over all $v \in X$:
$\sum_{v \in X} \sum_{u \in N^-(v)} f(u, v) = \sum_{v \in X} \sum_{w \in N^+(v)} f(v, w)$.
For all $e \subseteq X$, $f(e)$ appears once on each side; after cancelling those terms we're left with $f^+(X) = f^-(X)$.

Lemma 2: For all cuts (A, B), $f^+(A) - f^-(A) = f^-(B) - f^+(B)$.

Proof: We have shown that $v(f) = f^+(A) - f^-(A)$ because A and B are disjoint and $A \cup B = V$.

Lemma 3: Push(G, c,s,t, f , P) returns a new flow f', with value $v(f') = v(f) + C$ in $O(|V(G)|)$ time

Ford-Fulkerson:

Input : A (weakly connected) flow network (G, c, s, t).
Output : A flow f with no augmenting paths.

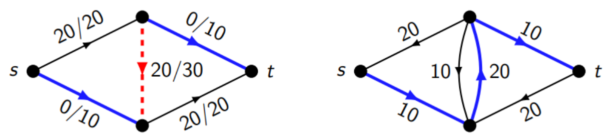
```
begin
    Construct the flow f with f(e) = 0 for all e ∈ E(G).
    Construct the residual graph G_f.
    while G_f contains a path P from s to t do
        Find P using depth-first (or breadth-first) search.
        Update f ← Push(G, c, s, t, f, P).
        Update G_f on the edges of P.
    Return f.
```

Complexity: Every step takes $O(|E|)$ time or $O(|V|)$ time, and since G is weakly connected we have $|V| = O(|E|)$. So the running time is $O(v(f*)|E|)$.

Flow Networks

Residual graph: G_f of (G, c, s, t) on V(G) as follows:

- if flow < capacity: then forward edge with value capacity-flow
- if flow > 0: add backward edge with value flow



Residual capacity of edge: max{capacity - flow, backward edge flow}

Residual capacity of network: minimum residual capacity of it's edges

Augmenting Path:

Max-flow min-cut theorem: The value of a maximum flow is equal to the minimum capacity of a cut, i.e. the minimum value of $c^+(A)$ over all cuts (A, B).

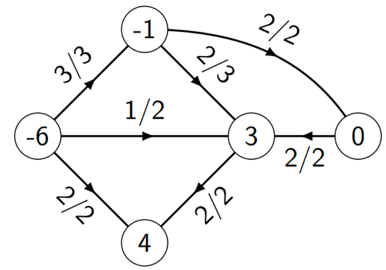
Proof: Let f be a maximum flow, and let (A, B) be a cut minimising $c^+(A)$. We already proved $v(f) \leq c^+(A)$. Moreover, there is no augmenting path for f , so exactly as before, there is a cut $(A'.B')$ with $c^+(A') = v(f)$; thus $v(f) \geq c^+(A)$. The result follows.

Special Flow Graphs

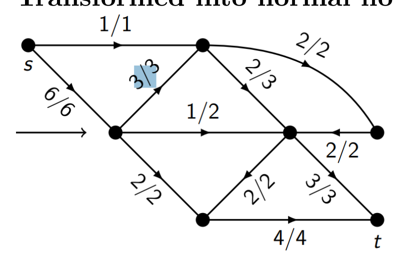
Circulation network: A circulation network (G, c, D) is a directed graph G = (V, E), a capacity function $c : E \to \mathbb{N}$, and a **Demand function** $D : V \to \mathbb{Z}$.

Circulation: A circulation is a function $f : E \to \mathbb{R}$ with $0 \leq f(e) \leq c(e)$ for all $e \in E$, and $f^-(v) - f^+(e) = D(v)$ for all $v \in V$.

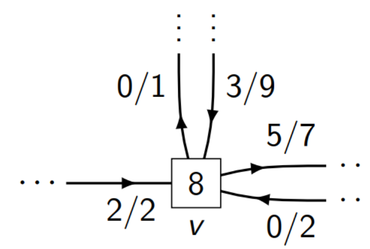
Demand Networks::



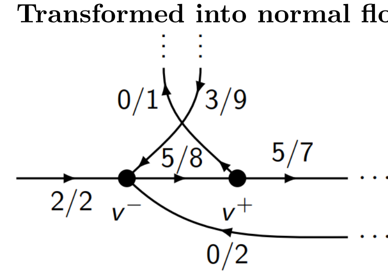
Transformed into normal flow graph:



Capacity flow:



Transformed into normal flow graph:



NP problems

NP: Formally, NP is the class of all decision problems X which have a polynomial-time algorithm **Verify** such that if and only if x is a Yes instance of X, then there is some bit string w (called a **witness**) with **Verify**(x,w) = Yes.

NP-hard: any problem in NP is Cook-reducible to it
NP-complete: is both NP-hard and in NP
CNF: And of Or clauses e.g. $(A \cup B) \cap (C \cup D)$
SAT problem: asks, "is this input satisfiable?". SAT is NP-complete.
Cook-levin Theorem: every problem in NP is reducable to SAT.
Cook reduction: from X to Y is an algorithm for problem X which, given an input of size s, runs in time poly(s) while making poly(s) calls to an oracle for Y whose input instances are all of size poly(s).
Oracle: for Y is a black box which, given an instance of problem Y , outputs a valid solution in O(1) time.

3-SAT: asks: is the input width-3 CNF formula satisfiable?
Theorem: 3-sat is np-complete
Proof:
C_i has width 2: Say $C_i = x \vee y$. Then we would like to replace C_i with $x \vee y \vee \texttt{False}$ in F', since this is True if and only if $x \vee y = \texttt{True}$.

But False is not a literal... Can we add a new variable which is always False in any satisfying assignment? Yes! If we add this CNF to F:

$$F_z = (\neg z_1 \vee z_2 \vee z_3) \wedge (\neg z_1 \vee z_2 \vee \neg z_3) \wedge (\neg z_1 \vee \neg z_2 \vee z_3) \wedge (\neg z_1 \vee \neg z_2 \vee \neg z_3)$$

then z_1 is forced to be False: No matter what value z_2 and z_3 take, their literals must both be False in one of the above OR clauses. ✓

If C_i has width 1: Say $C_i = \neg x$. Then we would like to replace C_i with $\neg x \vee \texttt{False} \vee \texttt{False}$... which we already know how to do!

We just need to introduce an extra copy of our always-False variable z_1 (since OR clauses can't contain two copies of the same literal). ✓

If C_i has width 3: We can just leave it as it is. ✓
If C_i has width $k \geq 4$: Say $C_i = \ell_1 \vee \cdots \vee \ell_k$. We would like to replace

$$C_i \to (e_1 = \ell_1 \vee \ell_2) \wedge (e_2 = e_1 \vee \ell_3) \wedge \cdots \wedge (e_{k-2} = e_{k-3} \vee \ell_{k-2}) \wedge (e_{k-2} \vee \ell_k),$$

as given the values of $\ell_1, \ldots, \ell_k$, this is satisfiable if and only if $\ell_1 \vee \cdots \vee \ell_k = \texttt{True}$. How do we implement the e_i's? We have

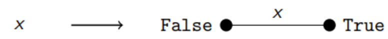
$$(a = b \vee c) \text{ if and only if } (a \vee \neg b) \wedge (a \vee \neg c) \wedge (\neg a \vee b \vee c);$$

the first two clauses on the right enforce $a = \texttt{False} \Rightarrow b \vee c = \texttt{False}$, and the last enforces $b \vee c = \texttt{False} \Rightarrow a = \texttt{False}$. □

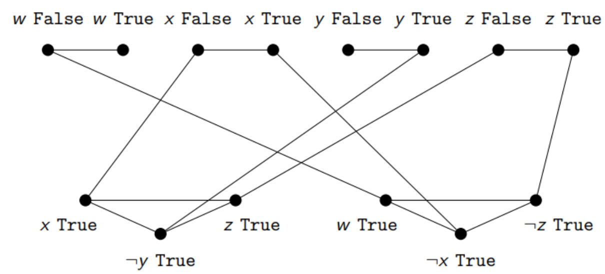
NP:

NP problems

Independant Set (IS): an independent set is a subset of V which contains no edges.
Decision problem example:
problem: what is the maximum independent set for graph G
Decision problem: Is there an independent set of size at least k for graph G
Theorem: IS is np-complete
Proof:
To simulate the variables of F, we want a gadget that can be in one of two states which will represent True and False...



So if $F = (x \vee \neg y \vee z) \wedge (w \vee \neg x \vee \neg z)$, say, how do we build G?



Vertex Cover (VC):
Theorem: VC is NP-complete.

We can verify a set is a vertex cover in polynomial time, so VC ∈ NP. We'll prove NP-hardness by proving IS $\leq_c$ VC.

This time though, we'll do it **non-constructively**, without gadgets.

Lemma: X is an independent set if and only if $V \setminus X$ is a vertex cover. (Because an edge intersects $V \setminus X$ if and only if it's **not** a subset of X.)

So G contains an independent set of size at **least** k if and only if G contains a vertex cover of size at **most** $|V| - k$.

Our reduction just passes the instance $(G, |V| - k)$ to our VC-oracle. □

SAT $\leq_c$ 3-SAT $\leq_c$ IS $\leq_c$ VC $\leq_c$ ILP -

Complement: Given a decision problem X, we write $\overline{X}$ for it's complement. Yes instances of X become No instances of $\overline{X}$ and vice-versa.
Co-NP: We define Co-NP to be the set of decision problems whose complements are in NP, such as $\overline{SAT}$
Karp Reduction vs Cook reduction:
$X \leq_C Y$ means "X is no harder than Y".
$X \leq_K Y$ means "X is a special case of Y."

Dynamic Programming

Dynamic programming design steps:
step 1: come up with an exponential-time recursive algorithm for your problem by reducing it to multiple smaller versions of itself
step 2: arrange things so that most of the calls of your recursive algorithm are repeated, and use this to make it polynomial
step 3: rewrite the algorithm as an iterative one (table) note: how can you collapse the recursive algorithm?
tips:
Think of your problem as a sequence of choices/ decisions
Recursively call program on each case that arises from the decision
Example:

Input : An array $\mathcal{R}$ of n requests and a weight function w.
Output : A maximum-weight compatible subset of $\mathcal{R}$.

begin
  if $\mathcal{R} = \emptyset$ **then**
    Return $\emptyset$.
  else
    Choose $I \in \mathcal{R}$ arbitrarily.
    Find the set X_I of intervals in $\mathcal{R}$ incompatible with I.
    $S_{\text{out}} \leftarrow \text{WIS}(\mathcal{R} \setminus \{I\}, w)$.
    $S_{\text{in}} \leftarrow \{I\} \cup \text{WIS}(\mathcal{R} \setminus X_I, w)$.
    if $w(S_{\text{out}}) > w(S_{\text{in}})$ **then**
      Return S_{out}.
    else
      Return S_{in}.

Memoise: every-time we do a call (maybe recursive), we store the result in a hashmap/ data structure
Why dijkstra doesn't work with negatives: dijkstra works locally, therefore doesn't work with negative weights
Bellman-ford recursive:

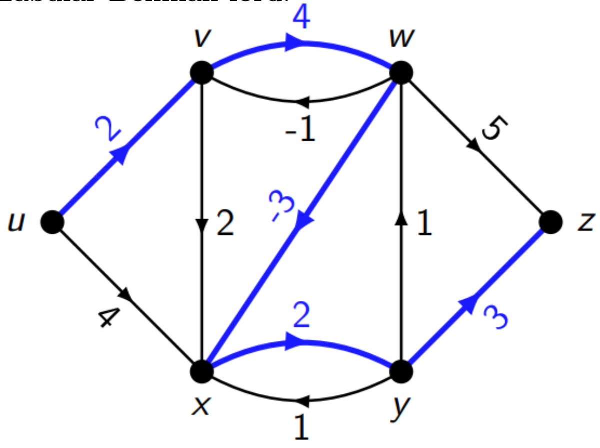
Input : A weighted digraph $G = ((V, E), w)$ with no negative-weight cycles, two vertices $s, t \in V(G)$, and an integer $k \geq 0$.
Output : A shortest walk from s to t in G with at most k edges, or None if none exists.

begin
  if $k = 0$ **then**
    Return the empty walk if $s = t$, and None otherwise.
  Write $N^+(s) = \{v_1, \ldots, v_d\}$, where $d \geq 1$.
  Let $P_i \leftarrow \textsc{GoodPath}(G, v_i, t, k-1)$ for all $i \in [d]$.
  if $P_i = \texttt{None}$ *for all* $i \in [d]$ **then**
    Return None.
  Return whichever walk is shortest in $\{sv_iP_i : i \in [d], P_i \neq \texttt{None}\}$.

Complexity: $|V|^2$ calls, each call is $O(|V|)$. $\therefore O(|V|^3)$

Dynamic Programming 2

Tabular Bellman-ford:



$k \backslash s$	u	v	w	x	y	z
5	$(uv, 8)$	$(vw, 6)$	$(wx, 2)$	$(xy, 5)$	$(yz, 3)$	$(z, 0)$
4	$(ux, 9)$	$(vw, 6)$	$(wx, 2)$	$(xy, 5)$	$(yz, 3)$	$(z, 0)$
3	$(ux, 9)$	$(vx, 7)$	$(wx, 2)$	$(xy, 5)$	$(yz, 3)$	$(z, 0)$
2	$(\emptyset, \infty)$	$(vw, 9)$	$(wz, 5)$	$(xy, 5)$	$(yz, 3)$	$(z, 0)$
1	$(\emptyset, \infty)$	$(\emptyset, \infty)$	$(wz, 5)$	$(\emptyset, \infty)$	$(yz, 3)$	$(z, 0)$
0	$(\emptyset, \infty)$	$(\emptyset, \infty)$	$(\emptyset, \infty)$	$(\emptyset, \infty)$	$(\emptyset, \infty)$	$(z, 0)$